



**MARCO
CANTÙ**

DELPHI 2010 HANDBOOK

Marco Cantù

**Delphi 2010
Handbook**

A Guide to the New Features of Delphi 2010; upgrading from Delphi 2009

Piacenza (Italy), February 2010

Author: Marco Cantù

Publisher: Wintech Italia Srl, Italy

Editor: Peter W A Wood

Tech Reviewers: Holger Flick, Daniele Teti, Marco Breveglieri, Chirs Bensen, Stefan Van As

Cover Designer: Fabrizio Schiavi

Copyright 2009-2010 Marco Cantù, Piacenza, Italy. World rights reserved.

The author created example code in this publication expressly for the free use by its readers. The source code for this book is copyrighted freeware, distributed via the web site <http://www.marcocantu.com>. The copyright prevents you from republishing the code in print media without permission. Readers are granted limited permission to use this code in their applications, as long as the code itself is not distributed, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning source code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, in the original or in a translated language, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Delphi is a trademark of Embarcadero Technologies. Windows Vista and Windows Seven are trademarks of Microsoft. Other trademarks are of the respective owners, as referenced in the text. The author and publisher have made their best efforts to prepare this book, and the content is based upon the final release of the software. The author and publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accepts no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

ISBN: 1450597262 (EAN-13: 9781450597265)

Delphi 2010 Handbook, First Edition, Revision 01.

Electronic edition licensed by Embarcadero Technologies, Inc. and sold by FastSpring and Plimus, on behalf of Wintech Italia Srl. Any other download or sale outlet is likely to be illegal. *This is not a free ebook, do not distribute it* (even if you received it for free from Embarcadero Technologies).

Printed copies of this book on sale on <http://www.amazon.com>.

More information and buying links on <http://www.marcocantu.com/dh2010>.

*Dedicated to my two wonderful kids,
Benedetta and Jacopo,
and their lovely mother, Lella*

4 - Dedication

Introduction

With the creation of the partially independent CodeGear business unit within Borland and the subsequent sale of the business unit to Embarcadero Technologies, Delphi has seen a significant increase in investment and is once again a growing and vibrant product thanks to its new technical features and to a developer community gaining in morale and affection, after a few years of slow growth and terms of capabilities and dwindling passion.

Embarcadero is investing more in Delphi than Borland did over almost the entire life of the product, and also improving the way it reaches out to the community. Long considered a “cash cow” with little future ahead of it, the product is now clearly at the center of Embarcadero's developer tools strategy, focused on native cross-platform development (which is going to be the direction of future versions of Delphi, according to the current product road map¹).

Delphi 2010 is another very significant step in this direction, after the impressive Delphi 2009 and a rather good Delphi 2007 release. From increased RTTI support to a significantly improved IDE; from the opening up to new databases (like Firebird) to the support of growing standards (like REST), Delphi 2010 is much more than an incremental new version. Its extended support for the Win32 platform, makes the latest Delphi the best tool, by far, for native devel-

1 The most recent Delphi road map, at the time of this writing, can be found at: <http://edn.embarcadero.com/article/39934>

6 - Introduction

opment for Windows 7. By devoting more than a couple of hundred pages to the new features of the product, this book is a testimony to the significant extension this version of Delphi offers to developers.

My Delphi Handbook Series

After a long series of Mastering Delphi books (published first through Sybex and then Wiley, when it acquired Sybex), over the last few years I've focused on specific books devoted to new features of individual versions of the product. The Delphi Handbook series doesn't cover Delphi from the ground up, but focuses only on new features.

By the time you are reading this, it should be possible to buy “*reprints*” of some of my classic Delphi books, along with buying my Mastering Delphi 7 or 2005 from online and traditional resellers. My basic offering is Essential Pascal².

Delphi 2007 Handbook, the first of my self-published volumes, covered new features from Delphi 7 to Delphi 2007, from IDE updates to language extensions, focusing on Windows Vista support and on the dbExpress data access library. This is the list of the chapters:

- The Delphi 2007 IDE
- Code Templates and Refactoring
- Project Management and MSBuild
- The Debugger
- Recent Updates to the Delphi Language
- Core RTL Changes
- Changes in the VCL
- Memory Management (and Robust Applications)
- Windows Vista and the VCL
- Database Support and dbExpress 4
- InstallAware and Other Tools
- Upgrading Projects to Delphi 2007

2 Essential Pascal is an introduction to the core features of the Pascal language. The focus is on traditional language structures and does *not* include object-oriented programming. More information at the book page: <http://www.marcocantu.com/epascal>

The **Delphi 2009 Handbook** had a long section on Unicode and delved into the significant changes to the language, which included generics and anonymous methods. There were also sections on the Ribbon user interface and the new DataSnap multi-tier architecture. This is the chapters list:

- What is Unicode?
- The Unicode String Type
- Porting to Unicode
- New IDE Features
- Generics
- Anonymous Methods
- More Language and RTL Changes
- VCL Improvements
- COM Support in Delphi 2009
- The Ribbon
- Datasets and dbExpress
- DataSnap 2009

The past two Delphi Handbooks are on sale in printed form both on Lulu and Amazon, while electronic versions can be bought online. Follow links on the book pages for buying printed or electronic versions:

<http://www.marcocantu.com/dh2007>
<http://www.marcocantu.com/dh2009>

The Delphi 2010 Handbook

The current book continues with this tradition by focusing on new features of Delphi 2010. Therefore, if you are upgrading from an older version of the product, you might want to read one or both previous handbooks first³.

There isn't a specific focus in this book, as there isn't one in Delphi 2010. The release brings to completion some of the recent features, like improved support for the Win32 API (with specific focus on Windows 7) and the new DataSnap architecture originally introduced in Delphi 2009 (now with HTTP support).

3 I might create a single all-encompassing Handbook Collection, but this still not a firm plan and it might not happen.

8 - Introduction

One of the new foundations of the product is its extended RTTI support and the inclusion of attributes in the Object Pascal language, the subject of one of the longest chapters. There was also a significant facelift in the IDE and debugger, with some easy to use features, and other more complex to understand and configure IDE extensions using the Delphi Open Tools API.

Needless to say the book covers all of this, and some more. Here is the list of the chapters, with more details available in the table of contents:

- 1. A Better IDE
- 2. The Debugger
- 3. Extended RTTI and Attributes
- 4. More and the Compiler and the RTL
- 5. The VCL and Windows 7
- 6. Touch and Gestures
- 7. Database Access and DataSnap
- 8. REST Web Services

The specific web page devoted to this book, including updates, source code downloads, and other information is at:

■ <http://www.marcocantu.com/dh2010>

Editor and Tech Reviewers

This book as seen the contribution of an editor and several tech reviewers, involved at various degrees, which provided a huge help and I won't be able to thank enough. The editor of this book (as of all my latest Delphi books) was Peter Wood, an IT professional who lives in Malaysia. I got technical feedback from Holger Flick, Marco Breveglieri, Stefan Van As, Daniele Teti, and Chris Bensen. Here is a short profile of each of them.

Daniele Teti

Daniele (<http://www.danieleteti.it>) is the R&D Director of bitTime Software, the Italian representative of Embarcadero. He is a passionate software developer and has been a speaker for Italian conferences on Delphi, PHP, design pattern, and multi-tier applications. Daniele has started a few open source project like the DataSnapFilterCompendium, a STOMP client, and a dependency injection framework for Delphi.

Marco Breveglieri

Marco (<http://www.marco.breveglieri.name>) in a long time Delphi programmer, trainer, and consultant, primarily involved in Microsoft Windows based software, targeting both the native and the .NET Framework platforms, and Web development using (X)HTML, CSS, JavaScript frameworks, and Microsoft ASP.NET MVC.

Chris Bensen

Chris (<http://chrisbensen.blogspot.com>) is a member of the Delphi R&D team who helped reviewing the chapter on touch and gestures, one of the areas of the product he worked on. He's also a great photographer.

Holger Flick

Holger (<http://www.flickdotnet.de/>) is a Delphi top developer and conference speaker, and it part of German's Delphi Expert team. Holger worked on Q&A for Embarcadero and has a deep knowledge of the product.

Stefan Van As

Stefan (<http://www.dutchdelphidude.com>) is a “Dutch Delphi Dude” and the current author of TopStyle4, a great HTML and CSS editing tool written in Delphi.

Author

I'm Marco Cantù, the author of this book. I've been in the “Delphi book writing” business ever since the first version of the product, when I released the original “Mastering Delphi” (a hefty tome of 1,500 pages). That was not my first writing experience, as I had previously written works on Borland C++ and the Object Windows Library.

The Mastering Delphi series, published by Sybex, was one of the best-selling Delphi book series for several years, with translations into many languages and sold in bookshops all over the world. More recently I started self-publishing the

10 - Introduction

Delphi Handbook series, available from multiple print-on-demand outlets, including Lulu and Amazon, and in PDF format.

Beside writing, I keep myself busy with consulting (mostly on applications architectures), help selling Delphi in Italy, do code reviews, Delphi mentoring, and general consulting for developers. I'm a frequent speaker at Delphi and general developer conferences (in Europe and in the Unites States), including the recent online *CodeRage* conferences organized by Embarcadero.

In 2009, Cary Jensen and I gave public training in both US and Europe at the jointly organized Delphi Developer Days, which are already planned for May 2010; for details (and future dates) see:

| <http://www.delphi.developerdays.com>

If you are interested in inviting me to speak at a public event or give a training session (on new Delphi features or any advanced Delphi subject) at your company location, feel free to send me a note by email.

Contact Information

To follow my activity you can use several online resources and communities.

In the following list you can see my blog (which I tend to keep quite active), my not-so-up-to-date personal site (a summary of my activities), my company site (with training offers), my Twitter account, and my Facebook page:

| <http://blog.marcocantu.com>
| <http://www.marcocantu.com>
| <http://www.wintech-italia.com>
| <http://twitter.com/marcocantu>
| <http://www.facebook.com/marcocantu>

I have an online mailing list based at Google groups. I also run an online newsgroup with a section devoted to discuss my books and their content. Here are the respective URLs:

| http://groups.google.com/group/marco_cantu
| <http://delphi.newswat.com/forumlistgroups?area=marcocantu>

Finally, feel free to drop me an email at my public address, although I generally don't offer tech support via email:

| marco.cantu@gmail.com

Table Of Contents

Introduction.....	5
My Delphi Handbook Series.....	6
The Delphi 2010 Handbook.....	7
Editor and Tech Reviewers.....	8
Author.....	9
Contact Information.....	10
Table of Contents.....	11
Chapter 1: A Better IDE.....	19
Installation.....	19
Proxy Configuration.....	20
Installation Folders.....	21
First Impressions.....	22
IDE Insight.....	23
Filter Wild Cards.....	24
Advanced: Customizing IDE Insight.....	25
The Delphi 2010 Editor.....	28
The Search Pane.....	29
Searching with Directory Groups.....	30
The Code Formatter.....	31
Live Templates and Code Completion.....	33
The Project Manager.....	34

12 - Table of Contents

Build All and Active Project.....	35
The Object Inspector.....	36
The Description Pane.....	36
The Component Editor Pane.....	37
Other IDE Features.....	38
Background Compilation.....	38
The Return of the Component Toolbar.....	39
Many More Recent Files.....	41
Use Unit Dialog.....	42
Updates to the Gallery.....	42
View Messages.....	43
What's Next.....	44
Chapter 2: The Debugger.....	47
Dragging the Instruction Pointer.....	47
Small UI Changes.....	49
Debugging Threads.....	50
Debugger Visualizers.....	53
Advanced: Visualizer Internals.....	55
Building a Value Replacer for UCS4Char.....	56
What's Next.....	59
Chapter 3: Extended RTTI and Attributes.....	63
Extended RTTI.....	64
A First Example.....	65
Compiler Generated Information.....	66
Larger Executable Files.....	67
The Rtti Unit.....	70
Rtti Objects Lifetime Management and the TRttiContext record.....	72
A Tree of Classes (and Class Information).....	74
RTTI for Packages.....	76
The TValue Structure.....	78
Reading a Property with TValue.....	80
Invoking Methods.....	80
Low-Level TValue.....	81
Custom Attributes.....	82
What is an Attribute?.....	83
Attribute Classes and Attribute Declarations.....	84
Browsing Attributes.....	86

RTTI Case Studies.....	88
Attributes for ID and Description.....	88
XML Streaming.....	93
What's Next.....	100
Chapter 4: More on the Compiler and the RTL.....	103
New Compiler Features.....	103
Version.....	104
Extracting Objects from Interface References.....	104
Class Constructors (and Destructors).....	106
Delayed Loading of DLL Functions.....	109
Scoped Enumerators.....	111
The With Statement Now Preserves Read Only Properties.....	111
New Run Time Library Features.....	113
RTL Trends.....	113
Browsing Existing Units.....	114
Collections and Containers.....	115
Discovering New Units.....	117
The Input/Output Utilities Unit.....	118
Extracting Subfolders.....	119
Searching Files.....	119
Filtering Sub-folders.....	121
Filtering Files.....	122
What's Next.....	122
Chapter 5: The VCL and Windows 7.....	125
Tech Overview of Windows 7.....	126
Delphi Support for Windows Vista.....	127
Notable Differences Between Vista and Windows 7.....	129
Delphi 2010 Windows API Units.....	131
New API Header Units.....	131
Extended Windows API Headers.....	133
Windows 7 Support.....	135
Working with Taskbar Buttons in Windows 7.....	135
Working with Libraries.....	140
DirectX for Forms.....	143
Direct2D.....	144
Gradients to the Max (With no Canvas).....	149
DirectWrite.....	151

14 - Table of Contents

Using the Windows Imaging Component.....	153
WIC Transformations.....	154
Other New VCL Features.....	156
Property Editors for Actions and Dates.....	157
Input Language and Language Libraries.....	158
Minor Incompatibilities with “Growing” Enumerations.....	159
What's Next.....	160
Chapter 6: Touch and Gestures.....	163
From Single Touch to Multi-Touch.....	164
Touch Hardware.....	165
Multi-Touch Pads.....	166
The Theory Behind Gestures.....	166
Towards a Touch-Based UI	167
The Gesture Manager of the VCL.....	168
A Basic Gesture Example.....	168
The Standard Gestures.....	171
Gestures and Actions.....	172
Custom Gestures.....	174
Database Gestures.....	179
Touch Keyboard.....	183
Multi-Touch Support.....	186
Handling wm_touch.....	186
Chris Bensen's TouchMove Demo.....	188
Inertia Manipulation (with no Touch).....	190
What's Next.....	196
Chapter 7: Database Access and DataSnap.....	199
New Field Types and Other Core Database Extensions.....	200
Themes Support and Other DBGrid Extensions.....	202
DBGrid In-place Editor Issues.....	203
Midas DLL Now With Source.....	206
ADO 2.8 Support.....	208
dbExpress in Delphi 2010.....	208
The Firebird Driver.....	208
Updated dbExpress Drivers: Interbase, MySQL, Oracle.....	210
The SQL Server Driver.....	210
DataSnap Updates.....	211
Overview of DataSnap in Delphi 2009.....	211

Overview of DataSnap in Delphi 2010.....	212
DataSnap over HTTP.....	213
A DataSnap HTTP Server with the Wizard.....	214
Testing the Connection in Data Explorer.....	216
HTTP Authentication.....	218
DataSnap WebBroker Integration.....	221
Overview of the WebBroker Architecture.....	222
The DataSnap WebBroker Wizard.....	223
A Client for the Web Server.....	227
Filtering Connections.....	228
Using ZlibCompression.....	229
Creating Custom Filters.....	231
JSON and Object Marshaling.....	233
Introducing JSON.....	233
JSON in Delphi 2010.....	234
Parsing JSON.....	236
Streaming Objects to JSON.....	237
Using JSON Converters and Reverters.....	240
JSON Values and Marshaling in DataSnap Server Methods.....	243
Server Methods Callbacks.....	247
The Server Side Implementation of a Callback.....	248
The Client Side Implementation of a Callback.....	249
What's Next.....	251
Chapter 8: REST Web Services.....	253
Why Web Services?.....	254
Web Service Technologies: SOAP vs. REST.....	254
XML and SOAP Updates.....	255
XML Processing in Delphi 2010.....	255
SOAP 1.2 Support.....	259
What is REST?.....	260
REST Architecture's Key Points.....	260
The REST Architecture and Delphi.....	261
REST Clients Written in Delphi.....	263
A REST Client for RSS Feeds.....	263
Of Maps and Locations.....	266
Google Translate API.....	270
Building a REST Server.....	274
An Echo Action.....	275

16 - Table of Contents

Returning the XML Data of a ClientDataSet.....	276
Returning a List of Customers.....	278
Building a DataSnap REST Server.....	281
Accessing the REST Server with a Browser.....	284
Returning Multiple Results.....	285
Calling the REST Server from a VCL Client.....	286
Calling the REST Server From a jQuery Client.....	288
Returning and Updating Objects with REST HTTP Methods.....	291
Listing Objects with a TJSONArray.....	295
Sending the List to the jQuery Web Client at Start-up.....	295
HTTP Methods: POST, PUT, and DELETE.....	298
Building a Database Oriented REST Server.....	302
REST Server Alternatives.....	305
What's Next.....	307
Index.....	309

Chapter 1: A Better IDE

The Integrated Development Environment (or IDE) is, for most developers, the key tool for writing applications with Delphi. The IDE in the 2010 version got a significant face lift, aimed at improving its overall usability. Rather than sporting incredible new features, the Delphi IDE lets developers perform many common tasks more easily and more quickly.

This chapter covers the main improvements of the IDE, without getting into too much detail, as in most cases it will be rather easy to pick them up. Still, there are less visible features and details you might easily miss which I'll try to cover.

Installation

As is true of the last few versions, the installation of 2010 Delphi is based on InstallAware. Installing the product is generally quite a smooth process, but there are a few elements worth mentioning.

The first relates to the requirements for the machines on which Delphi 2010 is being installed. As the IDE itself uses some .NET features, the presence of .NET 3.5 SP1⁴ has been added to the prerequisites. If you keep your Windows machine updated, you're likely to have already installed it.

Another change in the requirements is that Windows 2000 is no longer supported as a development platform, although it is still fully supported as a target operating system for running applications compiled in Delphi 2010. Support for running applications on Windows 9x was already dropped in Delphi 2009.

It is not that you absolutely cannot run the Delphi IDE on Windows 2000, but that Embarcadero Technologies gives you no guarantee it will work. In case you want to try to install Delphi 2010 on this operating system you can run the installation program with a specific flag⁵:

```
| Setup.exe /wi n2k
```

On a very positive note, even if this is really a minor issue, in the Delphi 2010 installer you can paste all four sections of the serial number at once, rather than having to paste each individual section.

Finally, consider that you can significantly reduce the installation footprint of the help system (and make the information much more appropriate to Delphi) if you disable the installation of the Microsoft Platform SDK, when installing the Delphi help. The details are in this blog post by Dee Elling:

```
| http://blogs.embarcadero.com/deeelling/2009/12/07/38310
```

Proxy Configuration

As an aside, there are two options for installing Delphi 2010 (and Embarcadero RAD Studio 2010). One is to buy or download the DVD with the complete version of the software. The second is to get the small footprint installer (the one

-
- 4 You might not be aware but .NET 3.5 SP1 provides countless improvements and is basically a brand new version of .NET compared to .NET 3.5. It has new libraries and features, not only bug fixes or limited changes. The reasons it was delivered as a patch were mostly commercial: to deliver it as an update and let more people download and install it on their machines, compared to a new version you must specifically decide to install.
 - 5 The win2k installation flag disables the check for Windows 2000 done by the installer, it doesn't change anything in the installation to make Delphi work on that version of the operating system. You can use it, but you won't be able to access technical support if anything goes wrong.

you'll generally receive when buying the Electronic Software Delivery (ESD) version of Delphi. This smaller installer will retrieve only the required installation files automatically based on your configuration and the edition you licensed.

There have been several reports about problems with this installer in the past, for developers behind a firewall and with a proxy configuration. What is not mentioned well enough is that the installer uses port 80 for downloading the installation files and that it uses the system wide proxy defined in the Internet Explorer configuration. So you shouldn't have any problems installing the ESD version even via a proxy server, providing you have Internet Explorer properly configured.

Installation Folders

For a long time Delphi was installed under the Program Files\Borland folder. With changes in the product ownership (first the Borland's CodeGear division and later Embarcadero Technologies) and the need to support Windows Vista folder permissions, the overall structure has kept changing considerably.

The main installation folder is now (by default):

```
| C:\Program Files\Embarcadero\RAD Studio\7.0
```

Other relevant folders include (on my computer, using the defaults) respectively projects, examples, database configuration, and sample database data:

```
| C:\Users\Marco\Documents\RAD Studio\Projects\  
| C:\Users\Public\Documents\RAD Studio\7.0  
| C:\Users\Public\Documents\RAD Studio\dbExpress\7.0  
| C:\Program Files\Common Files\CodeGear Shared
```

These folders are not very much different from the past two versions of the IDE, with the welcome addition of a sub-folder for the dbExpress configuration.

While the main installation folder has been changed from *CodeGear* to *Embarcadero*, the Registry settings are still saved under the more familiar:

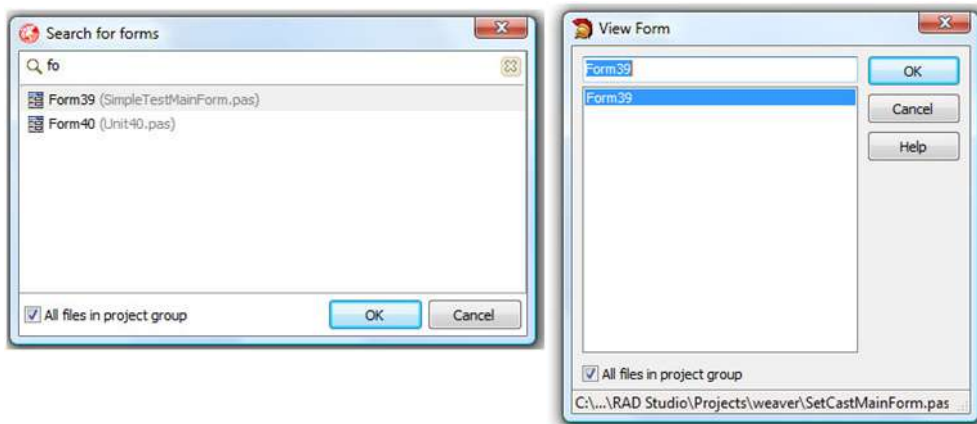
```
| HKEY_CURRENT_USER\Software\CodeGear\BDS\7.0
```

First Impressions



When you first start Delphi 2010, you won't see lots of big differences from Delphi 2009, but cleaner graphics, with new icons for the IDE and for your applications, by default. The new icon and style borrows heavily from the company style, but also revamps some of the classic elements of Delphi, like the three-column temple and the Greek helmet (shown up here). Needless to say you might like the new style or not, as it is mostly a matter of taste. I think it is a good step in the right direction, a more modern look without betraying the product history.

The overall user interface has been cleaned up somewhat, replacing some of the older dialog boxes of the product with versions that have a more modern user interface and (in many cases) extended search options. As an example, consider the View Form dialog box, now properly renamed Search for forms:



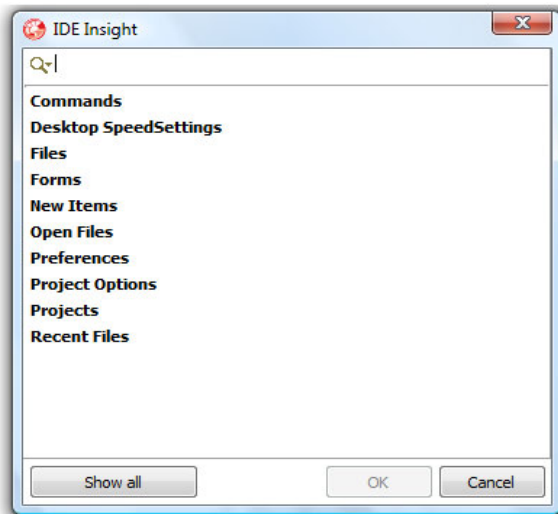
Not only does it have a nicer look, compared to the older one with the gray background, but it also has a very handy search capability, terribly useful for large projects. The View Units dialog box had been given the same kind of improvement. A similar makeover was made to the Use Units dialog box, covered at the end of this chapter. Oddly enough now that some of these dialog boxes have been cleaned up, most developers will use them less and less, simply because the information they list (like the forms and the units of the current project or the projects in the current project group) now shows up in the new IDE Insight dialog box, a sort of central starting point to find just about anything you might want to look for in the IDE and the current project.

IDE Insight

Both newcomers and expert users can easily get lost in the large number of menu items, settings, components, and features you can activate in the IDE. At times even experts get lost because features were moved from a version of Delphi they spent a lot of time with. That's why the team grabbed an idea that other development tools already implement and came up with searching capabilities in several dialog boxes (more on these later) and with an overall search mechanism for the entire IDE, called “IDE Insight”.

You activate this window by pressing the F6 key (or by using Ctrl + <period>)⁶. You can see the IDE Insight dialog box here on the right.

As you start typing into this dialog box, it will show a filtered list of just about anything you might want to look for in the IDE:



- **Commands** of the main menu of the IDE, including those added dynamically in the Tools menu or by Wizards or extensions of any kind (but the menu items of local popup menus)
- **Component Palette** elements, where the current view is a visual designer, like a form or a data module.
- **Components** used by the current designer, again where the current view is a visual designer. Components depend on the installed packages, and obviously include third-party ones.
- **Code Templates**, where the current view is an Object Pascal source code editor, a C++Builder editor, or any other editor supporting code templates.

⁶ You might have to press Alt-F1 if you are not using the default key bindings.

- **Desktop SpeedSetting**, usually managed with the corresponding toolbar of the main form, the one with the small combo box.
- **Files** include the list of files of the current project (and other projects in the group) and is available only if a project is active in the Project Manager.
- **Forms** filters the forms and designers of the current project, again only if a project is active.
- **New Items** has elements of the New Items dialog box.
- **Open Files** provides fast access to any file currently open in the editor.
- **Preferences** filters on the individual elements of the IDE preferences (the Tools | Options dialog box) and will open the corresponding page of the dialog box when selected.
- **Project Options** does the same with the options of the current project (again, you need to have a project open). Finding project options by typing their names is a superb feature I'm using a lot.
- **Projects** let's you jump to a project of the current project group.
- **Recent Files** and **Recent Projects** filter the recently closed source code files and projects (which in Delphi 2010 can be customized much more than in the past, as we'll see in the section "Many More Recent Files").

Notice that as you start searching, the IDE Insight dialog will show only a few elements of each category, unless you press the "Show all matches" button or the use corresponding Ctrl+E shortcut (which toggles between showing all categories with the best match in each one or showing all matches).

Filter Wild Cards

What is less intuitive to figure out is that you can use wild cards when typing in this search box (and most other search boxes available in the IDE):

- **?** will match any single character
- ***** will match zero, one, or more characters

Notice that an implicit ***** is automatically added both at the beginning and at the end of the search text to match sub strings. The same wild cards work in most of the other filtered search dialog boxes added to the IDE.

Advanced: Customizing IDE Insight

Warning: This is an advanced section using Delphi's Open Tools API. If you are not interested in extending the Delphi IDE or have never used such a low-level API, you might want to skip this section.

The list of elements in the IDE Insight is quite extensive, but there are certainly many others you might want to add to it. This is why the Delphi 2010 has a new Open Tools API (the extensions you can register with the IDE) specifically intended for adding custom IDE Insight entries. In this and the next chapter, I'll be covering a couple of specific IDE extensions written using the Open Tools API, but we have no room to discuss it from a broad perspective⁷.

Here I'm going to show you a trivial example, which has no particular goal other than adding an entry to that window. There are certainly several interesting ways to extend IDE Insight, from presenting existing Delphi files and projects to open (after searching some folders) to integrating more sophisticated bookmarks, from code search on web sites to hooking up help pages.

Although I don't have room for a detailed coverage of the architecture of Delphi's Open Tools API, here I'm presenting a demo that can get you started in creating your own IDE Insight extensions.

The service interface you have to call to interact with this portion of the IDE is called `IOtaIdeInsightService`. One of its methods, `AddItem`, lets you add an element to the IDE Insight window. You can do this when registering the given unit of your design time package:

```
var
  firstIdeInsight: IOtaIdeInsightItem;

procedure Register;
begin
  firstIdeInsight := TIdeInsightTest.Create('First Test');
  (BorlandIDEServices as IOtaIdeInsightService).AddItem(
    firstIdeInsight, 'Cantools');
end;
```

⁷ In the first few years of Delphi, there was some extensive documentation of the Open Tools API, which was originally way more limited than it is today. Now there are some online references, often not really up-to-date. Even some commonly used IDE extensions stick to low-level techniques rather than using newer portions of this API. Even though a good part of the API methods are commented in the `ToolsApi` unit, it is true that it is far from trivial to start using it. I have written papers on this topic in the past and presented it at conferences, but I've not been able to publish anything on the topic since Delphi 3!

26 - Chapter 1: A Better IDE

The code first creates an object of the `TIDEInsightTest` class, which implements the required `INTAIDEInsightItem` interface as you'll see shortly; second, it adds that item to a new section of the IDE Insight window (called '*Cantools*' after all of my IDE extensions).

There is further code in the unit to remove the IDE Insight item before the package is unloaded:

```
procedure RemoveIDEInsight;
begin
  (BorlandIDEServices as IOTAIDEInsightService).
    RemoveItem (firstIDEInsight);
  firstIDEInsight := nil; // it is an interface
end;

initialization

finalization
  RemoveIDEInsight;
```

The core of the code is in the `TIDEInsightTest` class, which implements the `INTAIDEInsightItem` interface and the basic `IOTANotifier` interface. The class is defined as follows, with the methods grouped by interface:

```
type
  TIDEInsightTest = class (TInterfacedObject,
    INTAIDEInsightItem, IOTANotifier)
  private
    FileName: string;
    Description: string;
  public
    constructor Create (const FName: string);

    { INTAIDEInsightItem }
    function DrawText(Canvas: TCanvas; Rect: TRect;
      var DrawDefault: Boolean; DoDraw: Boolean = True): Integer;
    procedure Execute;
    function GetDescription: string;
    function GetDescriptionSearchable: Boolean;
    function GetGlyph(Bitmap: TBitmap): Boolean;
    function GetSticky: Boolean;
    function GetTitle: string;
    function GetVisible: Boolean;
    procedure Update;
    { IOTANotifier }
    procedure AfterSave;
    procedure BeforeSave;
    procedure Destroyed;
    procedure Modified;
  end;
```

Most of the methods have a rather trivial implementation, with the constructor taking a string parameter that is later returned as the item title. The description, instead changes every time the Update method is called:

```

constructor TIDEInsightTest.Create(const aName: string);
begin
    fName := aName;
    fDescription := 'Sample IDE Insight entry';
end;

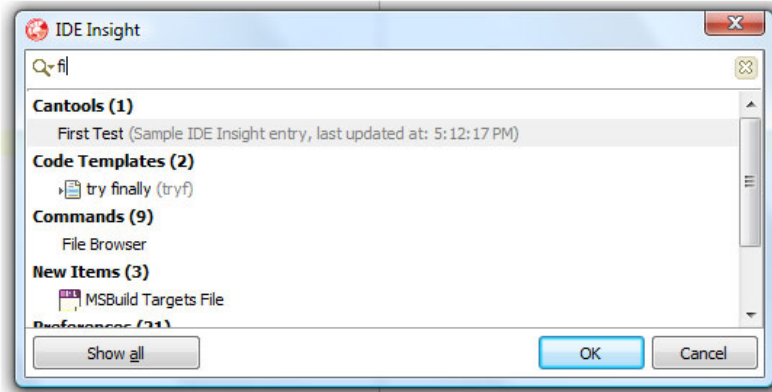
function TIDEInsightTest.GetTitle: string;
begin
    Result := fName;
end;

function TIDEInsightTest.GetDescription: string;
begin
    Result := fDescription;
end;

procedure TIDEInsightTest.Update;
begin
    fDescription := 'Sample IDE Insight entry, last updated at: ' +
        TimeToStr (Now);
end;

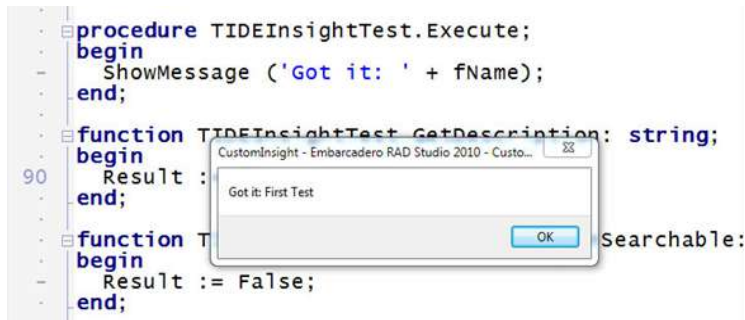
```

You can see the effect of these definitions in the image below, taken from the IDE Insight pane after installing the package with this custom extension:



As a user selects the new item, the Execute method IDE Insight object is called. In my case the implementation is not terribly interesting, but you can see the output along with the source code of its Execute method:

28 - Chapter 1: A Better IDE



Again, this is only an example showcasing the technology, not useful in itself. The ability to plug in custom actions in the IDE as you customize it beyond the planned activities (new components, new wizards, new Live Templates, new Tools configurations) is a significant change from the past. Let's hope open source communities and developers of Delphi paid add-on tools take advantage of this feature.

The Delphi 2010 Editor

The Delphi 2010 editor sees limited changes, some of which are interesting in terms of usability. Let me start with the most simple ones.

Since the early days of Delphi, and of its Turbo Pascal predecessor, developers using this language have got the habit of indenting their source code with two spaces rather than a tabulator, which is a more common approach in other languages. That's why the Tab key has been neglected for so much time, contrary to other development environments.

In the editor of Delphi 2010, if a code block (of one or more lines) is highlighted the Tab key will indent it, while pressing Shift+Tab will unindent it. This is exactly like using the Ctrl+Shift+I and Ctrl+Shift+U key combination that have been available for a long time. As you can see the only reason for this change is to help developers coming from other development environments or those who regularly have to use more than one.

Another small but nice feature is the ability to move editor bookmarks (which have been persistent between editing sessions since Delphi 2007) to different lines by dragging them in the gutter at the side of the editor. There is a specific

icon that gets displayed during this drag operation. As we'll see in the next chapter, a similar dragging operation is available also for breakpoints and (believe it or not) the instruction pointer.

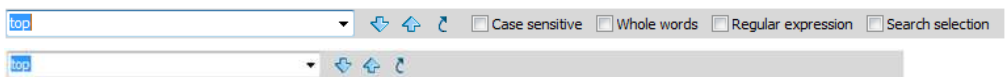
Among new options there is the ability to show all of the search matches (a new feature, discussed next) and to disable code folding (which was previously only available through a registry setting).

As you search for a term, all search terms found in the current file will be highlighted, using the color specified as “Additional search match highlight”⁸ (worth mentioning because I think you might really want to change the default, which I don't particularly like).

The Search Pane

In past versions of Delphi you could use the Ctrl+F combination (or the Search | Find menu command) to open a search dialog box, and afterward press F3 to search for the next occurrence of the term. A common alternative was to use the less powerful, but faster to use, Ctrl+E combination (invoking the Search | Incremental Search command), and then start typing in the editor status bar and jump to the searched element while typing.

Both Find and Incremental Search commands show specific search panes at the bottom of the editor, making the former easier to use but keeping the difference in the behavior. When you call Find you get the pane above but need to press Enter (or click on one of the arrow buttons) to activate the search; when you use Incremental Search you search as you type but have fewer options:



In both cases the number of matches is displayed and all elements found that are visible in the editor will be highlighted. Notice also that while you can use the arrow keys to move from a match to the next one and back, the classic F3 key and Shift + F3 keys still work. A related new feature is that as you get to the end of the file, Delphi will ask you to “Restart search from the beginning of the file” and let you make the “wrap around” setting permanent. (You'd later be able

8 In the Editor options | Colors page of the Options dialog box, this color is the last entry in the Elements list.

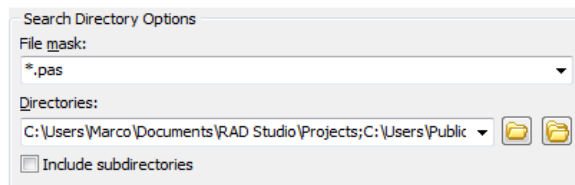
to change this setting in the main page of the Editor options.) In any case the wrap status is displayed at the right end of the pane, as shown here on the side.





Searching with Directory Groups

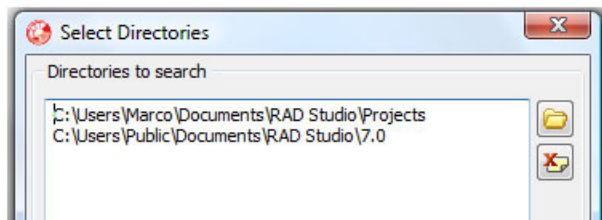
These improved file searching capabilities relate to finding information in the current file in the editor. For searching multiple files in the editor, in the current project, or in a given folder and its sub-folders, Delphi provides the Search | Find in Files command (or Ctrl+Shift+F).

This dialog box was made more flexible in Delphi 2010 by adding support for searching in multiple folders and for user-defined Directory Groups, a set of folders the user can refer to using a name. The find in Folders pane of the Find in Files Dialog box looks like on the side here.



In the Directories combo box you can have one or more folders, while in older versions of Delphi you could enter only one folder name. There is also a second button, with a new relevant behavior:

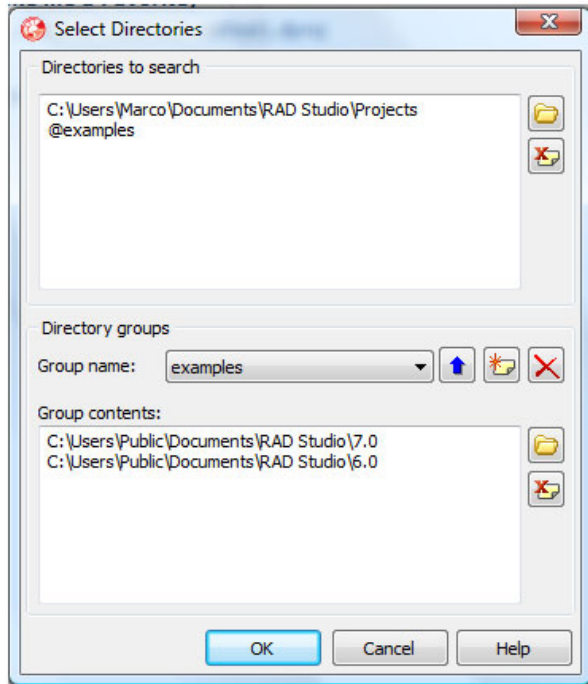
- The first button on the right (the one with a single folder) lets you pick one folder as in the past 
- The second button (the one with two folders), opens up a new Select Directories dialog, displayed below) 



You can now type the folder names, or pick a folder on each line with the upper button (with the folder), and cleanup invalid paths with the second button (the X over the page).



The bottom portion of the dialog can be used to give a name to a group of folders, which can later be used among the directories to search by prefixing the name with the @ sign. If in doubt, you can add a group to the list in the upper half of the dialog using the up arrow button (here on the right) that shows up as you select a group name in the combo box:



The actual line listed in the search dialog (which you can also type manually) would be like:

| C: \Users\Marco\Documents\RAD Studi o\Proj ects; @exampl es

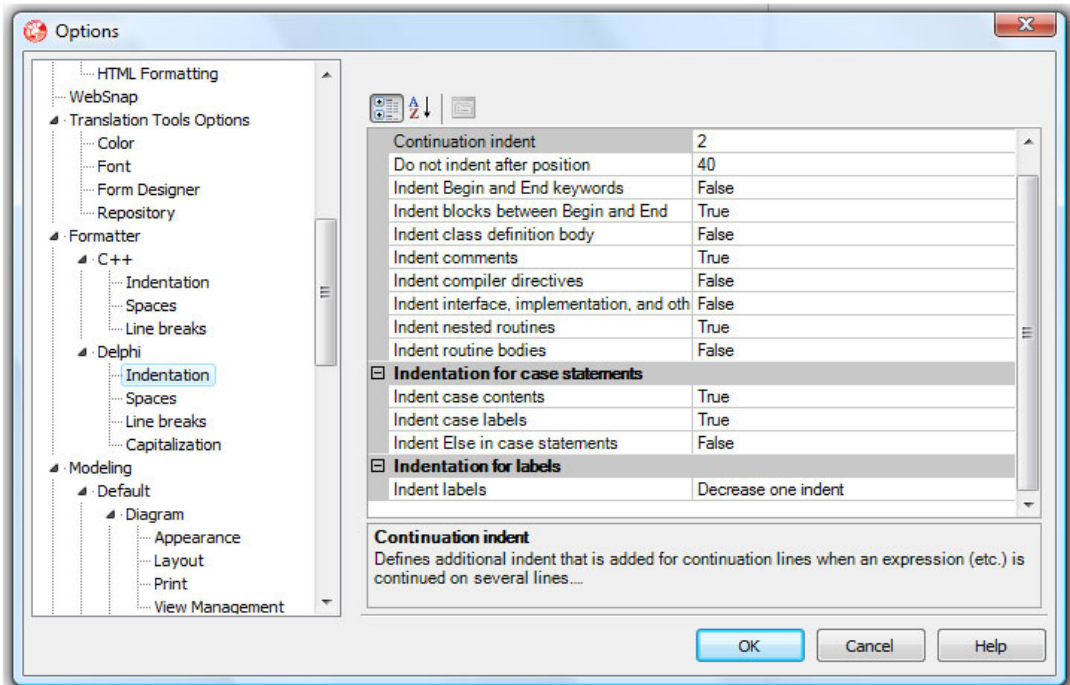
The Code Formatter

Many Delphi developers have long relied on third-party source code formatters to clean up the layout of existing code and promote company standards. Even if late to the game, Delphi itself now includes code formatting capabilities, with enough flexibility built into the system to make it worthwhile (even in what is clearly a first attempt, but still a good one).

32 - Chapter 1: A Better IDE

The Code Formatter is invoked with the Format Source command of the local menu of the editor or with the corresponding Ctrl+Alt+F shortcut. If there is some text selected in the editor, the formatting will be applied only to the selection, if not to the entire source code file. There doesn't seem to be a direct way to reformat the source code of all of the files in a project on opening the editor, although it might not be too difficult to write an IDE extension to accomplish this.

Despite the fact that there are some Delphi source code formatting guidelines, it is very difficult to find perfect agreement among different Delphi programmers on how exactly the code should be written. Formatting is generally subject of intense debate. The new source code formatter in Delphi 2010 provides 54 different options to fine tune its behavior. Granted this won't satisfy one million different formatting styles (one for each Delphi developer!), but it comes reasonably close.



One of the goals is at least to let developer format source code using the standard style used within the VCL source code and for the code generated at design time. This is almost the case with absolutely minimal exceptions, as long as you

keep the default settings. You can see some of the options in the screen shot in the previous page, but I won't cover each of them in detail as they are quite intuitive and the help section at the bottom of the pane is quite informative.

The proof of the quality of the code formatter would be in its actual use by the Delphi community, and although I anticipate this feature will find its strong detractors, I've had a limited but positive experience with it (using it to cleanup the source code for this book) and think that a fair number of developers will get used to it.

By the way, I find it quite odd that there isn't an option to apply the active formatting style to all files of a project. Doing this unit by unit can be extremely tedious.

Live Templates and Code Completion

There are some very minor improvements to refactorings and live templates, including three new live templates, in Delphi 2010. All of them are quite trivial. There is a `raise` template and a `todo` template, producing the following two lines of code respectively:

```
raise Excepti onType. Create(' Error');
{TODO -oOwner -cGeneral : ActionItem}
```

The third new live template is a variation of the class declaration, called `classf`, which produces the same code of the `class` template without comments in each section.

Speaking of refactorings, even if there are no brand new capabilities it is certainly worth noting that the Rename, Change Parameters, and Extract Method refactorings now work on generic types.

Finally, you can add the reserved words to Code Completion, by enabling the "Show reserved words" check box of the Editor Options | Code Insight page of the Options dialog box. This options becomes handy when you have to type some rather long reserved words like `initialization` or `resourcestrng`.

The Project Manager

There are enough small changes in the Project Manager to devote a small section to it. Being one of the most commonly used panes of the IDE, even minor changes in it become significant. Before we look into this, let me underline that project files (the new .dproj format used by MSBUILD and introduced in Delphi 2007) remain identical to Delphi 2009, to the point that they even carry the same version number 12.0. The actual version number of Delphi 2010, in fact, is 14⁹.

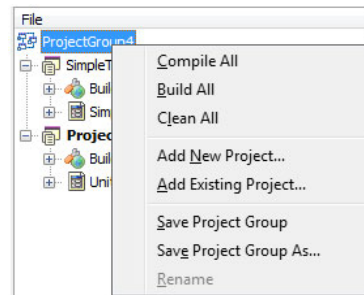
This is a snippet of the initial section of a new Delphi 2010 project file including the version information:

```
<Project xmlns="http://schemas.microsoft.com/...msbuild">
  <PropertyGroup>
    <ProjectGuid>{F7C0ED24-7449-...}</ProjectGuid>
    <ProjectVersion>12.0</ProjectVersion>
    <MainSource>SimpleTest.dpr</MainSource>
    <Config_Condition="'$(Config)'=='>Debug</Config>
    <DCC_DCCCompiler>DCC32</DCC_DCCCompiler>
  </PropertyGroup>
  ...
```

The fact that there was no change in the version number means you can directly open Delphi 2009 projects in 2010 and vice versa, with no conversions or any hidden changes.

After this brief introduction to the project files format, here are the new features of the Project Manager:

- Where there are multiple projects open in the Project Manager, the project group local menu will also have the Compile All, Build All, and Clean All commands, as shown here on the side. Individual project nodes have a new From Here menu item, with a sub-menu hosting the three Compile All From Here, Build All From Here, and Clean All From Here commands.



9 The reason the Delphi internal version number (not to be confused with the compiler version number, which is currently at 21.0) jumped from 12 to 14, is the same you won't find floor 13 in a US hotel or row 13 on a plane: that number is associated with bad luck!

- You can drag a file from the editor to the Project Manager window to add it to the current project.
- The local menu of a project has a new Sort By menu that let's you sort the project's units by Name, Modified Date, Type, or Path. If you keep the Auto Sort option on, the setting will be applied to any new unit added to the project and in case status of a unit changes. If Auto Sort is off, the sorting option you enable will be applied only once. In case you manually reorder one of the units of the project, the Auto Sort options will be turned off.
- The Project Manager toolbar has a sort button that let's you define the sort order for all of the open projects and also preset some default sort options.
- In case of package projects, there is a new Uninstall menu. Notice that you don't generally need to use this command, as when you Compile or Build a package, if this is already installed it will be automatically uninstalled before compiling it and reinstalled after the compilation. In specific circumstances, though, it is handy to remove a package directly from the Project Manager.
- Notice also that depending whether the package is installed or not its icon in the Project Manager changes, with the small gear turning from gray (not installed) to yellow (installed). This icon with the gear is used by design time packages, while run time packages have the base icon with no gear (they cannot be installed in the IDE).
- By the way, you can now also select multiple packages in the Project Manager and perform an Install operation on all of them at once, using the local menu of the Project Manager.



Build All and Active Project

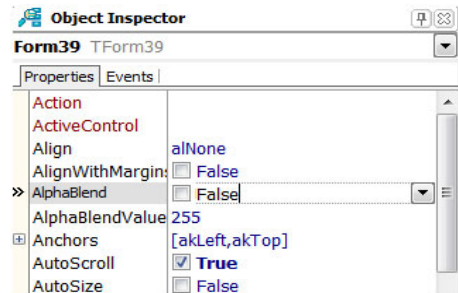
There is another significant change (which I don't like really) in the way the project manager behaves when building multiple projects. What used to happen when doing a Build All was that each project would become the active project in turn and you could use this visual clue to see the progress of a long compilation. Now you get information about each project while it compiles, but no overall view.

Another difference, and possibly the reason for this change, is that at the end of the multi-project compilation the current project doesn't change. If you debug a project, do a rebuild all, and then start debugging again you'll be working on the same project, regardless of their sequence. This is actually quite nice, but I wish the two effects (display the current project being compiled, plus restoring

the one you were working on at the end of the compilation) could be combined, rather than adding one at the expense of the other.

The Object Inspector

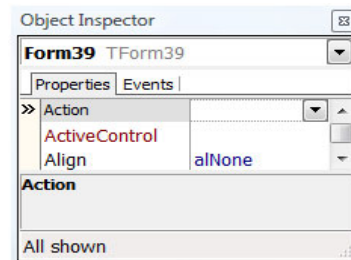
Even if the changes to the Object Inspector in Delphi 2010 are limited, they are certainly worth a look, considering that this is a window developers generally spend a lot of time working with. The first noticeable change is the new property editor for Boolean values, which displays a check box you can use to toggle the value (although the drop down list with True and False is still available).



The Description Pane

Two more changes are visible at the bottom of the Object Inspector. First, there is now a Description pane at the bottom of the Object Inspector. This pane (originally introduced in the IDE during the development of Delphi for .NET) is supposed to show information about the current property, but all it does is repeat the property name. This seems quite a waste: You can reduce its size to a single line (or even less than a line), but there is no obvious way to remove it altogether.

Since I really don't like it, I've written a small *informal*¹⁰ IDE plugin to get rid of it. The unit that removes that pane is part of the CustomInsight package. In its Register



10 An *informal* plug-in of the Delphi IDE, in my personal jargon, is one that doesn't use the Open Tools API but manipulates IDE object directly, something that can easily be done considering the Delphi is itself a VCL application, so you can reach for the global Application and Screen objects and ... look around.

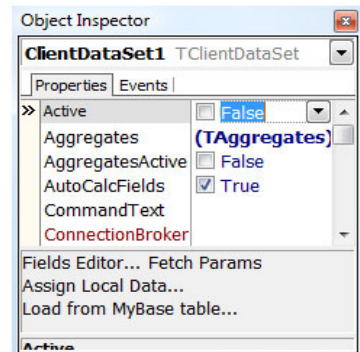
procedure it looks for the Object Inspector windows by scanning the Screen. Forms list, finds the component called *DescriptionPane*, and sets its Height property to zero. The skeleton of the code (without some of the needed tests) is the following:

```
function FindObjectInspector: TComponent;
var
  I: Integer;
begin
  Result := nil;
  for I := 0 to Screen.FormCount - 1 do
    if Screen.Forms[I].ClassName =
      'TPropertyInspector' then
      Exit (Screen.Forms[I]);
  end;

  procedure Register;
  var
    aComp: TComponent;
  begin
    aComp := FindObjectInspector;
    aComp := aComp.FindComponent('DescriptionPane');
    (aComp as TControl).Height := 0;
  end;
```

The Component Editor Pane

The second new pane is an area devoted to the component editor menu items, following the style used by Visual Studio. Traditionally in Delphi component editors (that is, special actions you can perform on given components at design time) showed up in the local menu of the designer, once the component was selected. Now, while the commands in the local menu are still there, the same information is visible in another pane at the bottom of the Object Inspector, as you can see here on the side for the ClientDataSet component¹¹. This new user interface makes the component editor commands easier to reach not only for developers using Visual Studio, but even for most existing Delphi



11 Notice that in this case I've removed the Object Inspector status bar and reduced the Description pane to the minimum, without using the IDE plug-in covered earlier.

users, as you often could only use the component local menu to figure out if a editor was available, while now this is clearly visible once a component has been selected.

Notice you can now select the component with the combo box of the Object Inspector and activate its Component editors, without having to make it visible in a designer as it was the case until Delphi 2009. The Return of Component Toolbar

Other IDE Features

Beside the new IDE Insight capability and the new features that relate with the editor, there are other changes that are worth highlighting, from background compilations to the Components toolbar.

Background Compilation

While in the past compiling a large application would force you to stop using the IDE while the operation was taking place, in Delphi 2010 you can turn on background compilation. This is not meant to speed up the compilation, but only to avoid blocking the IDE while doing so.

Considering Delphi compilation speed, this is hardly noticeable for the average application, but when compiling a large project group that takes some time it certainly can be nice to keep working while Delphi is compiling. When you turn on this setting, keep in mind that Delphi will take a sort of snapshot of the files in the editor during a background compilation, so that the changes you perform after issuing a compilation request won't be seen by the compiler.

How can you take advantage of this feature? An example would be looking at the source code files for warnings the compiler has already issued, while it keeps compiling. Another options could be adding breakpoints. You can also edit files, but this won't make a lot of sense if you plan debugging your program after you've compiled it. In fact, when you issue a Run (or F9) command, background compilation won't be used, regardless of your settings. The same happens for a Step Over (F8) command.

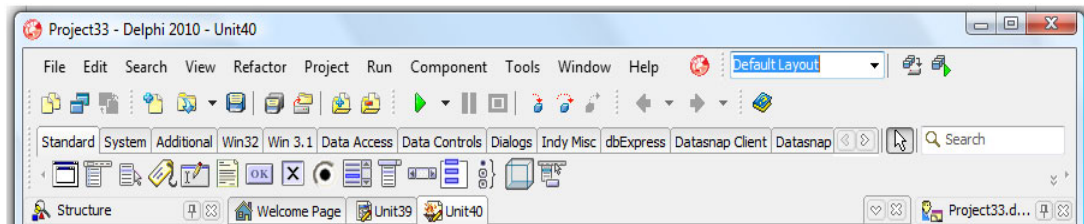
Still, running a compilation while you keep writing more code will reduce following compile times and (at times) improve the reliability of the Code Insight information, like Error Insight.

In any case there is a long list of other operations you cannot do during a background compilation, from changing compiler options to closing a project or activating another one, from using any refactoring to executing another compilation, from installing packages to starting the debugger. The full list is available in the Delphi 2010 help file under:

ms-help: //embarcadero.rs2010/rad/Background_Compilation.html

The Return of the Component Toolbar

In Delphi 2010 there is now an option to display the components visible in the Tool Palette in a toolbar below the main menu¹². This is extremely similar to how the Component Toolbar used to look like in Delphi 7 and before), and the feature has been specifically added as a way to convince some of the Delphi 7 aficionados to migrate to a new version:



You can actually go to some extreme and make Delphi 2010 look almost exactly like Delphi 7. This was covered in a short video by Andreano Lanusse of Embarcadero Technologies at:

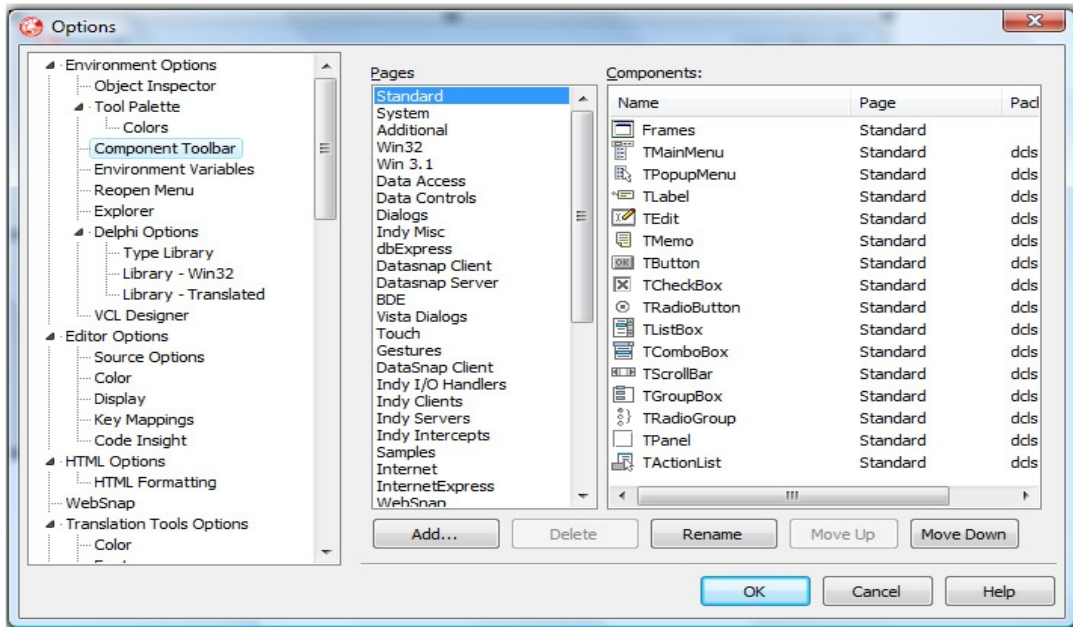
<http://blogs.embarcadero.com/andreanolanusse/how-to-configure-delphi-2010-to-look-work-and-feel-like-delphi-7/>

The new toolbar, visible in the image above, has the classic tabs to pick on of the pages plus a local menu with the tabs in alphabetic order¹³ and a Search box active like a component filter (removing pages with no matching components).

¹² The code of the new Component Toolbar was originally part of the “Andy’s DDevExtensions” and was donated to the product by Andreas Hausladen.

You can drag the tabs to reorder them (they'll start by matching the sequence in the Tools Palette).

There is also a specific page of the Options dialog that you can use to fully customize and configure the Component Toolbar, as you can see below. Notice that if you change the Component Toolbar configuration, this won't stay in sync any more with the Tools Palette, which is the default behavior.

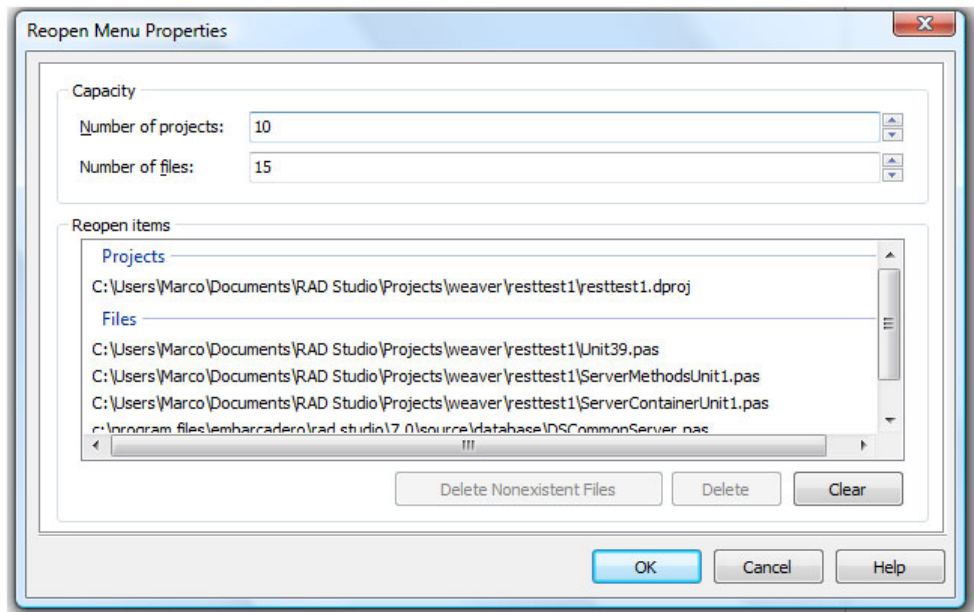


Speaking of the Tools palette, you can now rename Palette categories (like those hosting the components) using a new corresponding menu. The Palette has another new feature: it remembers the current position (with the current open categories) when switching designers.

-
- 13 In case you are wondering what the Win 3.1 tab is for, its hosts a set of very old VCL controls that pre-date Windows 95 and have been kept around for compatibility reasons. Nothing to do with the development of 16-bit applications for Win 3.1, a feature available only in the first version of Delphi, many years ago.

Many More Recent Files

In Delphi the lists of recent projects and files have always been limited to a fixed size, 5 for projects and 10 for files (units). Now in the File | Reopen menu you can pick the Properties command and open up a configuration dialog box, which lets you both change that number and and clean up the list, by removing non-existing files and individual entries you don't care about:



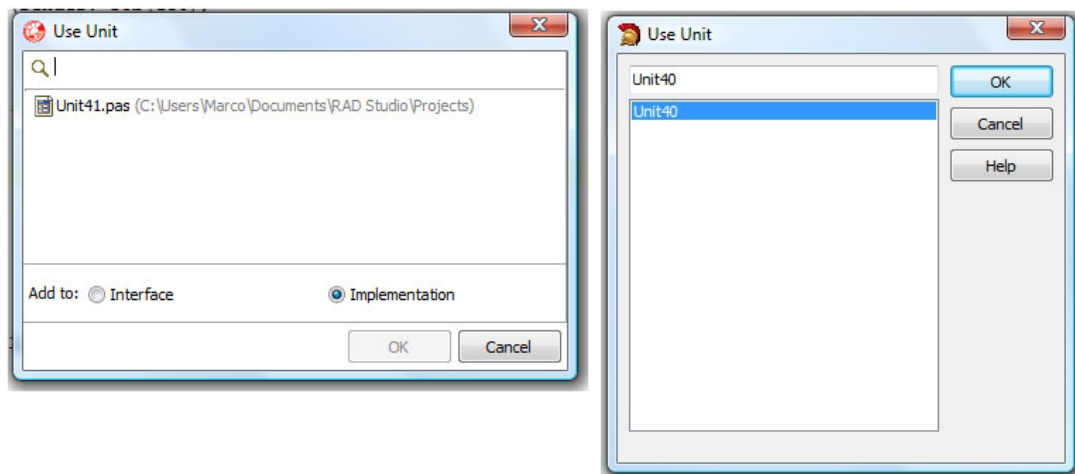
Although this is a nice addition, I think I'd still prefer using the project management features of the Welcome page, which let's you pick specific projects and mark them as favorites, so that they remain available over time. In the Welcome page you can also group favorite projects and manage these categories. If you are used to that (like I am), the extensions to the recent files will help you only with individual files and units, not with projects. Though using both techniques together gives the best of both worlds.

Keep in mind, though, that the recent files can be stored on a project by project basis, when you let Delphi save Project Desktop settings. In this case the list of recent files depends on the active project, which is not a bad idea after all. That list, though, cannot be managed in the same way of the global one.

There is a problem that might happen if you increase the number or files or projects and your screen has a limited resolution. In case the second level menu with the list of projects and files doesn't fit on the screen, it will simply not show up at all (rather than showing a partial list)! This is a bug of the `PopupMenu` control used by the IDE and the only work around is to reduce the number of items in the list.

Use Unit Dialog

Among the dialog boxes that have been made to look more modern, there is certainly the Use Unit dialog box, displayed below in both the Delphi 2010 and “classic” Delphi 2009 versions. This dialog box also shows a small new feature, the ability to add the unit you are interested in to the `uses` statement of the `interface` section of the current unit or to the `uses` statement of the `implementation` section (which was the only option in the past). Again, you can now search a unit with the filter at the top of the dialog box.

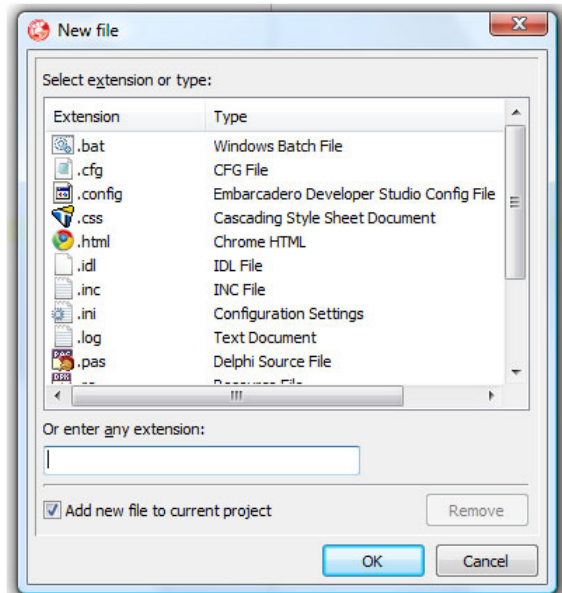


Updates to the Gallery

The last dialog box that I want to mention is the Gallery, or New Items dialog box. As many other windows, it now has a search box to filter its content (content that also shows up in IDE Insight), but also has some changes in its behavior.

Rather than showing only the available options for the current context, as happened in the past, the Gallery now shows all of the possible new items: those not available for the current context are grayed out. This makes it easier for users to find things, and also to figure out what is currently not available. In the past, at times one had to go through many pages looking for something that was simply not there because it was not available for the current context. It used to be quite annoying.

Another update to the New Items dialog box relates to the selection of a new text file (Other Files | Text File). Rather than simply opening a new blank text file, the Delphi IDE now asks for an extension and whether you want to add the file to the project. This way adding an .INI file, an XML file, or other configuration file to your projects becomes easier. The New file dialog box has the long list of extensions partially visible in the image on the side.



The list of extensions can be modified using the same dialog box and is stored in the Registry using the *Exts* key under:

|| HKEY_CURRENT_USER\Software\CodeGear\BDS\7.0\NewFileDlg

View Messages

The View menu has a new command, View | Messages, that lets you open the message pane at the bottom of the editor. When you compile a program that has errors or warnings, that pane is opened automatically. In case there are no errors, you might still want to look at the information about the compilation added to the Output tab (with the command line executed for compiling) or the Build tab, like the following:

|| Checking project dependencies...

44 - Chapter 1: A Better IDE

```
Compiling Project33.dproj (Debug configuration)
Success
Elapsed time: 00:00:01.2
```

Oddly enough, since this output was added in Delphi 2007, there was no way to see it for a successful compilation (unless the Message pane was already open because of a previous compilation error).

What's Next

In this chapter I focused on the new features of the IDE and the editor of Delphi 2010, but I actually skipped a significant area of the IDE, the debugger. The reason is that there are quite a few interesting new features for the debugger, so I wanted to devote its own chapter to this specific topic.

In Chapter 2, I won't simply look at new features of the debugger but will also cover new debugger customizations.

Chapter 2: The Debugger

One area of the Delphi Integrated Development Environment that has seen a significant number of improvements is the Integrated Debugger. This is one of the reasons why I decided to devote a specific chapter to this topic even if it is a short one.

The second reason is that the Delphi Debugger is more powerful than most developers realize and it often an underused area of the product, with many little known features, and by covering it specifically I hope to reverse this trend.

I'll start the chapter with a new nifty feature, dragging the instruction pointer, look at various UI and functional changes, and spend the final part of the chapter focusing on the new debugger visualizers.

Dragging the Instruction Pointer

We have seen in the last chapter that in gutter of the Delphi editor you can now drag bookmarks and also breakpoints. This can be useful, but is not that

48 - Chapter 2: The Debugger

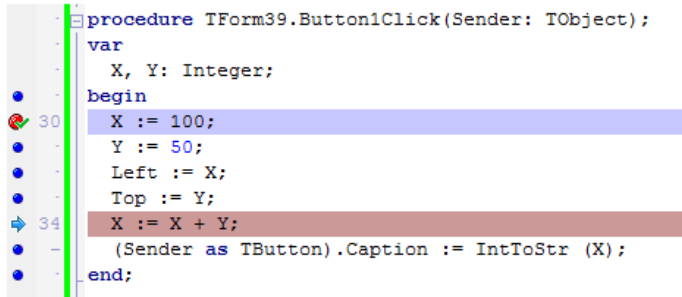
important, as creating a new bookmark or thread is generally a matter of a click (although a bookmark with complex rules will take some time to configure).

A similar dragging operation is also available (believe it or not) for the instruction pointer (or IP). This means that while you are debugging an application, you can move the instruction pointer forward, skipping some statements or move it back, executing a statement a second time. In between, of course, you can alter a local variable or perform other operations as usual.

Let me illustrate this new feature, that I find extremely powerful, with an example. Suppose you have this code (which is part of the MoveIP demo);

```
procedure TForm39.Button1Click(Sender: TObject);
var
  X, Y: Integer;
begin
  X := 100;
  Y := 50;
  Left := X;
  Top := Y;
  X := X + Y;
  (Sender as TButton).Caption := IntToStr (X);
end;
```

If you place a breakpoint on the first line, and start stepping into the statements (pressing F8), you can get to the line in which you add X and Y, as in the following image:

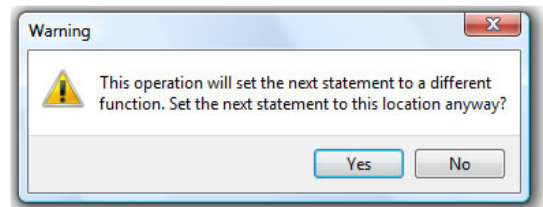


Now you can press F8 again, to execute that line, move the mouse over the instruction pointer (the left-to-right light blue arrow), drag it back to the addition, and repeat the process a few times. This way you can execute the same statement multiple times, increasing the value of X. In more practical situations you can repeat an operation after changing some variables rather than having to stop the program and restart debugging. In real world situations getting back to a given method can take quite some time.

Another option would be to place the same breakpoint, execute the initial assignment of X, move the instruction pointer to the addition (skipping the initialization of Y and the property assignments), to see what it happens in the program. At times, it might be very handy to skip some operations and test the following ones.

Just for fun, you can also go back to the statement adding X and Y, execute it, move the instruction pointer back to the original assignment of X, then bump it to the final line, and display 100 in the button's caption.

In theory you can even drag the instruction pointer to a different method, in which case the debugger will show you the warning displayed here on the right side. In most cases, though, this will result in an access violation as the specific code is not properly initialized and cannot be executed.



Again, being able to change the execution flow of a program without having to stop the debugger, actually edit the code, recompile the program, and restart debugging getting the program again in the same status is an extremely significant step forward for the debugger. In my view, this is one of the most useful new features of Delphi 2010, but it seems that it was somewhat neglected in terms of documentation and product marketing.

Small UI Changes

In general, Delphi 2010 sees quite a few changes in terms of the debugger user interface in addition to its new capabilities. Here I'm providing just a very short overview. The Event Log view uses an optimized Virtual Tree View component and adds support for multi-line output from `OutputDebugString` calls and exception messages. In the same pane, when auto scrolling is enabled, you can temporarily disable scrolling by clicking on the pane, and resume it by selecting the last item (pressing the End key).

In the local menus of the Local Variables view, the Watches view, and the Debug Inspector pane, there are new commands for opening the

Evaluate/Modify dialog box for the given symbol or expression and for creating a new watch.

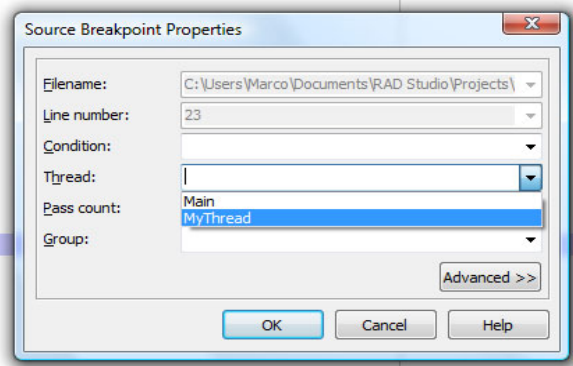
There are changes also to the CPU view, with the option of “following” registers, but details are relevant only to those who know how to use this debugger view (which are a very few, myself excluded). Another small but interesting change is that the debugger tries to remember the *expanded* status (that is, if you've expanded any of the properties or ancestor objects) of watched variables and variables in the Local view. Next time you stop at a breakpoint, the previous situation will be restored if possible.¹⁴

Debugging Threads

If debugging applications can at times become a very complex task, debugging multi-threaded applications is always complex and can become a daunting activity. The debugger, in fact, interferes with the flow of execution of the threads, for example slowing the thread that is being stepped into. Also while you are debugging a method you've placed a breakpoint in, other threads might need to call the same method, causing a great deal of confusion.

To help you out a little bit (making things a little more manageable, even if still very complex) the debugger in Delphi 2010 introduces a few thread-oriented operations.

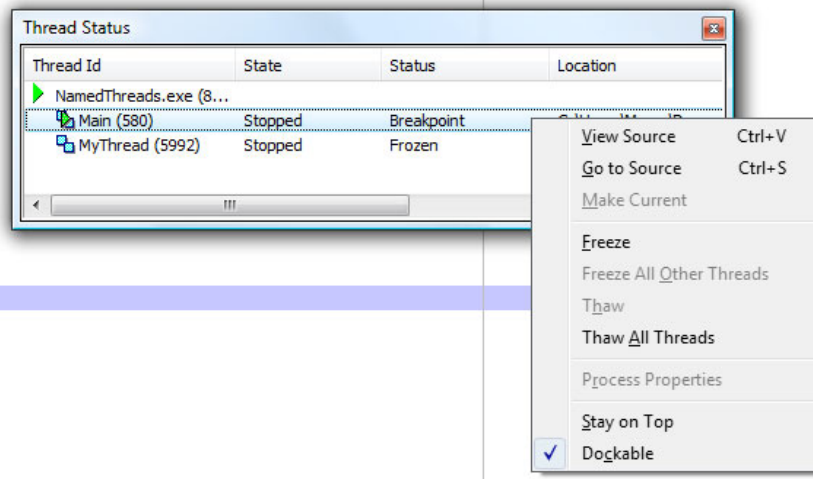
First, when you set a breakpoint you can tie it to a specific thread. This is particularly handy when placing a breakpoint in a method or routine executed by multiple threads. If you name the threads, you'll see a list with the names of the active threads, as in the image,



¹⁴ More information about the fix to this long-standing issue in Chris Hesik blog at: <http://blogs.embarcadero.com/chrisbesik/2009/10/06/34989>

which will remain valid for following debug sessions. If the threads have no name, you'll get the IDs of the threads being executed. The option to specify a given thread for a breakpoint is available for Source Breakpoints (the commonly used ones), the Address Breakpoints, and also for Data Breakpoints.

Once a thread (the main thread or a secondary one) is stopped in the debugger, you can take control of the threads' execution. In the Thread Status View pane, in fact, you can use new local menu commands for *freezing* and *thawing* (that is, *unfreezing*) a given thread or all threads except the current one:

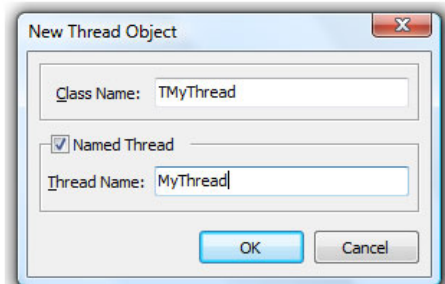


As you can see in the previous image, the status of the thread named MyThread is now *frozen*, after calling the Freeze command for the given thread.

For experimenting with threads in the debugger I've create a basic application that uses named threads, called Named-Threads.

If the ability to create named threads was already in Delphi 2009, the code generated by the Thread Wizard of the New Items dialog box is considerably different. In Delphi 2009 the code of your named thread will include the following:

```
procedure TMyThread.SetName;
var
```



52 - Chapter 2: The Debugger

```
ThreadNameInfo: TThreadNameInfo;  
begin  
  ThreadNameInfo.FType := $1000;  
  ThreadNameInfo.FName := 'MyThread';  
  ThreadNameInfo.FThreadID := $FFFFFFFF;  
  ThreadNameInfo.FFlags := 0;  
  
  try  
    RaiseException( $406D1388, 0,  
      sizeof(ThreadNameInfo) div sizeof(LongWord),  
      @ThreadNameInfo );  
  except  
  end;  
end;  
  
procedure TMyThread.Execute;  
begin  
  SetName;  
  { Place thread code here }  
end;
```

Now the same code, in Delphi 2010, is part of a new class method of the TThread class, called NameThreadForDebugging, so the same code becomes:

```
procedure TMyThread.Execute;  
begin  
  NameThreadForDebugging('MyThread');  
  { Place thread code here }  
end;
```

The same class method can also be called from the main thread, to give it a thread name like “Main”. Again, doing so let's you refer to the main thread in a standard way between consecutive debug sessions.

Getting back to the NamedThreads example, it has a thread that computes a string with the current time by calling a trivial global function:

```
function CurrentTimeAsStr: string;  
begin  
  Result := TimeToStr (Now);  
end;
```

This single line of code is called by both the secondary thread, in its own context, and by the main thread. Calling a synchronized method of the main form would have defeated the example altogether, as I need a single function called in two different threads to demonstrate placing a thread-specific breakpoint.

The thread calls the function in a loop that keep going until its Terminate flag is set and uses an anonymous method for synchronizing (again calling the CurrentTimeAsStr function within the synchronized code would have defeated the purpose of this example):

```

procedure TMyThread. Execute;
var
  strTime: String;
begin
  NameThreadForDebugging('MyThread');
  FreeOnTerminate := True;

  while not Terminated do
    begin
      sleep (1000);
      strTime := CurrentTimeAsStr;
      Synchronize(procedure ()
        begin
          FormNamedThreads. Log ('MyThread: ' + strTime);
        end)
    end;
end;

```

The global function `CurrentTimeAsStr` is called also by the main thread in the handler of an `OnTimer` event:

```

procedure TFormNamedThreads. Timer1Timer(Sender: TObject);
begin
  Log ('Main: ' + CurrentTimeAsStr);
end;

```

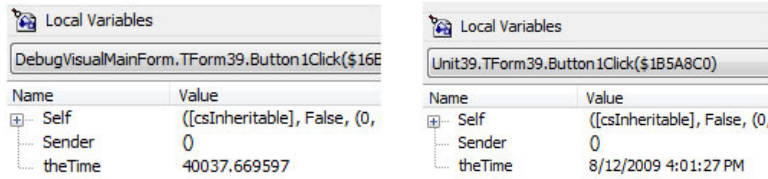
Other than that, the main form has code for creating and freeing a single instance of the thread object. You can experiment with this project by placing a breakpoint in the `CurrentTimeAsStr` function and try stepping into the first thread that stops, first with no freezing and later freezing the other thread. Notice that if you freeze the main thread, the `Synchronize` call won't return (because it needs to execute code in the context of the main thread, which is blocked), effectively blocking also the secondary thread. In this case (with no thread blocked in the debugger), you cannot thaw the frozen thread. What you can do, instead, is pause the program: at that point the Thread Status View will let you issue the various freezing and thawing commands.

Debugger Visualizers

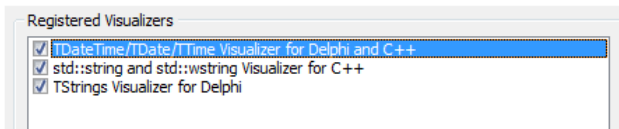
Another useful change in the architecture of the Delphi debugger is the ability to plug in specific visualizers for complex data structures. For example in the past when looking at a `TDateTime` variable, you'd see its internal floating point value. In Delphi 2010, instead, one of the pre-installed visualizers lets you see the date and time value in a specific and more meaningful way. This is quite

54 - Chapter 2: The Debugger

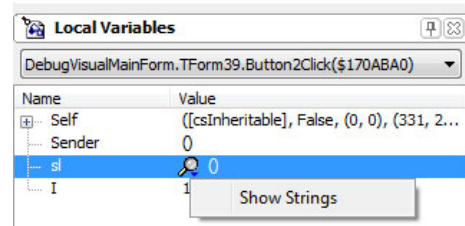
obvious if you compare the Delphi 2009 view of a local `TDateTime` variable on the left with the Delphi 2010 one on the right:



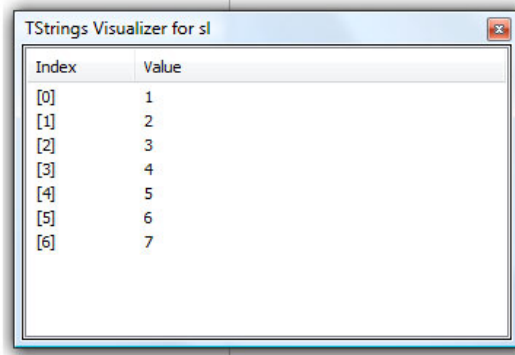
You can see this visualizer by experimenting with the `DebugVisual` project of the current chapter. There aren't many such visualizers pre-installed in the Delphi IDE. One of them is the *TDateTime Visualizer*, and the other is a *TStrings Visualizer*, as you can see in the new page of the Options dialog (Debugger Options | Visualizers) showing the visualizers configuration:



The good news, however, is that third parties can build and plug-in their own custom visualizers into the IDE. You can do the same for any specific data structures you tend to work with. The two predefined visualizers are examples of the two different kinds of visualizers available in Delphi. A **value replacer**, like the *TDateTime Visualizer* displayed earlier, shows its value directly in place of the default one the debugger would display (including the Evaluator Tooltips, the Watch View, the Locals View, the Evaluate/Modify dialog, and the Debug Inspector View. An **external viewer**, like the *TStrings Visualizer* shown in action here on the right, adds to the default debugger representation a magnifying glass icon, which brings up a menu with a view command.



Activating this command opens up a separate window, which can generally be docked alongside the others panes of the IDE. This is the viewer of the *TStringsVisualizer*, the only one external viewer supplied with Delphi 2010:



Advanced: Visualizer Internals

Warning: This and the next are two advanced sections using the Open Tools API. If you are not interested in extending the Delphi IDE or have never used such a low-level API, you might want to skip the remainder of this chapter.

With only two visualizers available, it is quite clear that what is important is the plug-in ability, and the possibility for a power user to configure this feature more than what's already in the box (and also to distribute its visualizers just like other IDE extensions). This is why it is worth looking behind the scenes, opening up another new area of the Open Tools API, just like I did in Chapter 1 regarding customization of the IDE Insights.

This time less effort is needed, because Delphi 2010 ships with the complete source code of its two ready-to-use visualizers, which provide good blue prints for experiments. The code is in the `Win32\Visualizers` section of the source code that comes with the Delphi installation (under `Embarcadero\RAD Studio\7.0\source`).

If you look in the `ToolsAPI` unit (the unit hosting most of the Open Tools API interface), you can see that the `LOTADebuggerServices` interface has a new method for installing a visualizer in the IDE:

```
procedure RegisterDebugVisualizer(
const Visualizer: IOTADebuggerVisualizer);
```

The parameter is an object implementing the `IOTADebuggerVisualizer` interface, which provides the overall information about the visualizer like its unique name, a description, and the data types to which it can be applied. The visualizer object must also implement one of the two specific interfaces for the two types of visualizers I mentioned earlier:

56 - Chapter 2: The Debugger

```
IOTADebuggerVisualizerValueReplacer  
IOTADebuggerVisualizerExternalViewer
```

The first has a method receiving a string with the value to be evaluated and returning a different one, after some custom processing; the second has a method to let you add an entry to the type inspection menu item and a second method executed when the menu item is invoked. You can have only one value replacer for each data type, while you can have multiple external viewers.

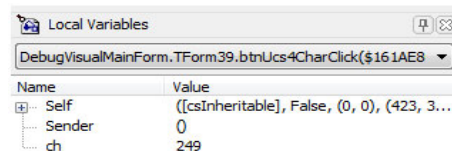
Building an external viewer is more complex, as you'll have to integrate the user interface with the IDE, providing docking support by implementing the other Open Tools API interface, and provide a way to refresh the output depending on the status of the debugger. You obtain this integration by implementing the `IOTADebuggerVisualizerExternalViewerUpdater` interface, which has four rather complex methods. That's why here I'll build a value replacer¹⁵.

Building a Value Replacer for UCS4Char

Before we delve into the development of a debugger visualizer, let me point you to a specific situation in which the debugger provides limited information about the values of a type. Consider the following code snippet, which is part of the `DebugVisual` example which is also used to demonstrate the other debugger visualizers.

```
procedure TForm39.btnUcs4CharClick(Sender: TObject);  
var  
  ch: UCS4Char;  
begin  
  ch := Ord('ù');  
  ShowMessage(Character.ConvertFromUtf32(ch));  
end;
```

If you put a breakpoint in the code above and look to the value of `ch`, you'll see its numerical value, not the character it represents, as shown on the right. It would be nice to see the accented letter, something we can now do thanks to this new feature.



¹⁵ Refer to the source code of the *TStringsVisualizer* for an example of the other type (which is certainly more interesting, but also quite complex).

To install a custom visualizer we have to create a design time package that requires the `designde.dcp` package and add a unit with a class implementing two debugger visualizer interfaces (the base one and the specific one) plus the `LOTAThreadNotifier` interface.

The class must provide all of the methods (here grouped by interface), even if most of them will have an empty implementation:

```

type
  TDebuggerUcs4CharVisualizer = class (
    TInterfacedObject, LOTADebuggerVisualizer,
    LOTADebuggerVisualizerValueReplacer, LOTAThreadNotifier)
  public
    { LOTADebuggerVisualizer }
    function GetSupportedTypeCount: Integer;
    procedure GetSupportedType(Index: Integer;
      var TypeName: string; var AllDescendants: Boolean);
    function GetVisualizerIdentifier: string;
    function GetVisualizerName: string;
    function GetVisualizerDescription: string;

    { LOTADebuggerVisualizerValueReplacer }
    function GetReplacementValue(const Expression,
      TypeName, EvalResult: string): string;

    { LOTAThreadNotifier }
    procedure EvaluateComplete(const ExprStr: string;
      const ResultStr: string; CanModify: Boolean;
      ResultAddress: Cardinal; ResultSize: Cardinal;
      ReturnCode: Integer);
    procedure ModifyComplete(const ExprStr: string;
      const ResultStr: string; ReturnCode: Integer);
    procedure ThreadNotify(Reason: TOTANotifyReason);
    procedure AfterSave;
    procedure BeforeSave;
    procedure Destroyed;
    procedure Modified;
  end;

```

The methods implementing the `LOTADebuggerVisualizer` interface provide information about the debugger visualizer, including the name and description that will be displayed in the Debugger Options | Visualizers page of the Options dialog box, covered earlier:

```

function TDebuggerUcs4CharVisualizer.
  GetVisualizerIdentifier: string;
begin
  Result := ClassName;
end;

function TDebuggerUcs4CharVisualizer.
  GetVisualizerName: string;
begin

```

58 - Chapter 2: The Debugger

```
    Result := 'Ucs4Char Visualizer for Delphi';  
end;  
  
function TDebuggerUcs4CharVisualizer.  
    GetVisualizerDescription: string;  
begin  
    Result := 'Displays the Unicode string for a Ucs4Char';  
end;
```

The last two methods return the number of types the visualizer can be used for, and each of their names (in this case, there is only one type, making the code much simpler):

```
function TDebuggerUcs4CharVisualizer.  
    GetSupportedTypeCount: Integer;  
begin  
    Result := 1;  
end;  
  
procedure TDebuggerUcs4CharVisualizer.GetSupportedType(  
    Index: Integer; var TypeName: string;  
    var AllDescendants: Boolean);  
begin  
    AllDescendants := False;  
    TypeName := 'UCS4Char';  
end;
```

The other relevant method is `GetReplacementValue` of the specific interface for this type of visualizer, `IOTADebuggerVisualizerValueReplacer`:

```
function TDebuggerUcs4CharVisualizer.GetReplacementValue(  
    const Expression, TypeName, EvalResult: string): string;  
var  
    ch: UCS4Char;  
begin  
    ch := StrToIntDef (EvalResult, 0);  
    Result := Character.ConvertFromUtf32 (ch);  
end;
```

In this specific example, all of the `IOTAThreadNotifier` methods have an empty body (but if you don't refer to that interface, the IDE will raise a low-level exception, so it is compulsory to have those empty methods).

Now that we have a class implementing our debugger visualizer for the `UCS4Char` type, we can install it after creating a global object, and clean up when the package is unloaded:

```
var  
    Ucs4CharVis: IOTADebuggerVisualizer;  
  
procedure Register;  
var  
    DebuggerServices: IOTADebuggerServices;  
begin
```

```

    Ucs4CharVi s := TDebuggerUcs4CharVi sual i zer. Create;
    if Supports(BorI andI DEServi ces, IOTADebuggerServi ces,
        DebuggerServi ces) then
        DebuggerServi ces. Regi sterDebugVi sual i zer(Ucs4CharVi s);
end;

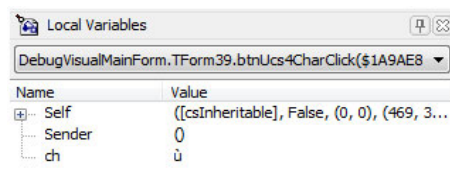
procedure RemoveVi sual i zer;
var
    DebuggerServi ces: IOTADebuggerServi ces;
begin
    if Supports(BorI andI DEServi ces, IOTADebuggerServi ces,
        DebuggerServi ces) then
        begin
            DebuggerServi ces. Unregi sterDebugVi sual i zer(
                Ucs4CharVi s);
            FreeAndNi l (Ucs4CharVi s);
        end;
end;

i ni ti al i zati on
fi nal i zati on
    RemoveVi sual i zer;

```

After compiling the package with this unit and installing it, if you debug the same sample program again you'll see the Local Variables view here on the right, including the accented letter

assigned to the Ucs4Char variable. This is much more informative than the original version¹⁶. It shows the value of installing similar in-place viewers for any non trivial low-level data types or structures you are working with.



What's Next

In the first two chapters I've provided you with a complete overview of the new IDE features in Delphi 2010. It is now time to move to another important topic, that is the changes in the Object Pascal compiler. The most significant new feature of the compiler in Delphi 2010 is the support for Extended RTTI and Attributes, a topic to which I've devoted the entire Chapter 3. In the chapter after that I'll touch on other interesting, even if less ground-breaking features of the compiler and also focus on changes to the Run Time Library.

¹⁶ This visualizer will only work for Unicode characters that the default font can display.

Chapter 3:

Extended RTTI

And Attributes

The area of the compiler that has seen the most significant update in Delphi 2010 is the generation and management of Run Time Type Information, RTTI. Traditionally, compilers of strongly, statically typed languages, such as Pascal, provided little or no information about the available types at runtime. All the information about data types was visible only during the compilation phase.

The first version of Delphi broke with this tradition, by providing run time information for properties and other class members marked with a specific compiler directive, `published`. This feature was enabled for classes compiled with a specific setting `{$M+}` and is the foundation of the streaming mechanism behind DFM files and the way you work with the form and other visual designers. When it was first made available in Delphi 1, this feature was a completely new idea, and along the years other environment and development tools adopted and extended it in several ways.

First, there were extensions to the type system (available only in Delphi) to account for method discovery and dynamic invocation in COM. This is supported in Delphi by dispatch ID, applying methods to variants, and other COM-related features. Eventually COM support in Delphi was extended with its own flavor of run time type information.

The advent of managed environments, such as Java and .NET, brought forward a very extensive form of run time type information, with detailed RTTI bound by the compiler to the executable modules and available for discovery by programs using those modules. This has the drawback of unveiling some of the program internals and of increasing the size of the modules, but it brings along new programming models that combine some of the flexibility of dynamic languages with the solid structure and the speed of strongly types ones.

Whether you like it or not (and this is indeed the subject of intense debate) Delphi is slowly moving into the same direction, and the adoption of an extensive form of RTTI in Delphi 2010 marks a very significant step in that direction. As we'll see, you can opt out of the new RTTI, but if you don't you can leverage some extra power in your applications.

The topic is far from simple, so I will proceed in steps. We'll first focus on the new extended RTTI that's built into the compiler and the new classes of the `Rtti` unit that you can use to explore it. Second, I'll look at the new `TValue` structure and dynamic invocation. Third, I'll introduce custom attributes, a feature that parallels its .NET counterpart and let's you extend the RTTI information generated by the compiler. Only in the last part of the chapter will I try to get back at the reasons behind the extended RTTI and look at practical examples of its use.

Extended RTTI

The compiler in Delphi 2010 generates by default much more extended RTTI information than any past version. Run time information includes all types, including classes and all other user defined types as well as the core data types predefined by the compiler and covers published fields as well as public ones, even protected and private elements. This is needed to be able to delve into the internal structure of any object.

A First Example

Before we look into the information generated by the compiler and the various techniques for accessing them, let me jump towards the conclusion and show you what can be done using RTTI. The specific example is very minimal and could have been written with the older RTTI, but it should give you an idea of what I'm talking about (also considering that not all Delphi developers used the traditional RTTI explicitly).

Suppose you have a form with a button, like in the `RttiIntro` example. You can write the following code to read the value of the control's `Caption` property:

```
uses
  Rtti;

procedure TFormRttiIntro.btnInfoClick(Sender: TObject);
var
  context: TRttiContext;
begin
  Log(context.
    GetType(TButton).
    GetProperty('Caption').
    GetValue(Sender).ToString);
end;
```

The code uses the `TRttiContext` record to refer to information about the `TButton` type, from this type information to the RTTI data about a property, and this property data is used to refer to the actual value of the property, which is converted into a string. If you are wondering how this works, keep reading. My point here is that this approach can now be used not only to access a property dynamically, but also to read the values of fields, including private fields.

We can also change the value of a property, as the second button of the `RttiIntro` example shows:

```
procedure TFormRttiIntro.btnChangeClick(Sender: TObject);
var
  context: TRttiContext;
  aProp: TRttiProperty;
begin
  aProp := context.GetType(TButton).GetProperty('Caption');
  aProp.SetValue(btnChange, StringOfChar(
    '*', random(10) + 1));
end;
```

This code replaces the `Caption` with a random number of `*`s. The difference from the code above is that it has a temporary local variable referring to the RTTI information for the property. Now that you have an idea what we are into,

let's start from the beginning by checking the extended RTTI information generated by the compiler in Delphi 2010.

Compiler Generated Information

There is nothing you have to do to let the compiler add this extra information to your executable program (whatever its kind: application, library, package...). Just open a project and compile it. By default, the compiler generates Extended RTTI for all fields (including private ones) and for public and published methods and properties. You might be surprised by the fact that you get RTTI information for private fields, but this is required for dynamic operations like binary object serialization and tracing objects on the heap.

You can control the Extended RTTI generation according to a matrix of settings: On one axis you have the visibility and on the other the kind of member. The following table depicts the default:

	Field	Method	Property
Private	x		
Protected	x		
Public	x	x	x
Published	x	x	x

Technically, the four visibility settings are indicated by using the following set type, declared in the System unit:

```
type
  TVisibilityClasses = set of (vcPrivate,
                               vcProtected, vcPublic, vcPublished);
```

There are some ready to use constant values for this set indicating the default RTTI visibility settings applied to TObject and inherited by all other classes by default:

```
const
  DefaultMethodRttiVisibility = [vcPublic, vcPublished];
  DefaultFieldRttiVisibility = [vcPrivate, vcPublished];
  DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

The information produced by the compiler is controlled by a new directive, \$RTTI, which has a status indicating if the setting is for the given type or also for its descendants (EXPLICIT or INHERITED) followed by three specifiers to

set the visibility for methods, fields, and properties. The default applied in the System unit is:

```
{ $RTTI INHERIT
  METHODS(DefaultMethodRttiVisibility)
  FIELDS(DefaultFieldRttiVisibility)
  PROPERTIES(DefaultPropertyRttiVisibility) }
```

To completely disable the generation of extended RTTI for all of the members of your classes you can use the following directive¹⁷:

```
{ $RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([]) }
```

When using this setting, consider it will be applied only to your code and a complete removal is not possible, as the RTTI information for the RTL and VCL classes is already compiled into the corresponding DCUs and packages. Keep also in mind that the `$RTTI` directive doesn't cause any change on the traditional RTTI generated for published types: This is still produced regardless of the `$RTTI` directive¹⁸.

What you can do with this directive is avoid the Extended RTTI being generated for your own classes. At the opposite end of the scale, you can also increase the amount of RTTI being generated, including private and protected methods and properties, if you wish (although it doesn't make a lot of sense).

Larger Executable Files

The obvious effect of adding Extended RTTI information to an executable file is that the file will grow larger (which has the main drawback of a larger file to distribute, as the extra loading time and memory footprint would be almost unnoticeable). For very small programs, the effect is somewhat minimal, but in large applications the increase will be significant. I'll look at very small programs first, testing two applications I've used over the years and with multiple versions of Delphi, to see how their size changes. After that, we'll look at a more real-world situation.

-
- 17 You cannot place the RTTI directive before the unit declaration, as it happens for other compiler directives, because it depends on settings defined in the System unit. If you do so, you'll receive an internal error message, which is not particularly intuitive. In any case, just move it after the unit statement.
 - 18 The new RTTI processing classes, available in the Rtti unit covered in the coming section, hook to the traditional RTTI and its `PTypeInfo` structure.

68 - Chapter 3: Extended RTTI and Attributes

The MiniSize program is not an attempt to build the smallest possible program, but rather to build a very small program that does something: it reports the size of its own executable file. This program doesn't use the VCL and only a very minimal portion of the RTL. All of the example code is as follows:

```
program MiniSize;

uses
  Windows;

{$R *.RES}

var
  nSize: Integer;
  hFile: THandle;
  strSize: String;

begin
  // open the current file and read the size
  hFile := CreateFile (PChar (ParamStr (0)),
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
  nSize := GetFileSize (hFile, nil);
  CloseHandle (hFile);

  // copy the size to a string and show it
  SetLength (strSize, 20);
  Str (nSize, strSize);
  MessageBox (0, PChar (strSize), 'Mini Program', MB_OK);
end.
```

The program opens its own executable file after retrieving its name from the first command-line parameter (`ParamStr (0)`), extracts the size, converts it into a string using the old-fashioned `Str` function, and shows the result in a message. The program uses the `Str` function for the integer-to-string conversion to avoid including the `SysUtils` unit, which defines all of the more complex formatting routines and would impose a little extra overhead. The following list compares past versions with Delphi 2010:

- Delphi 5 compiled this program to 18,432 bytes
- Delphi 6 reduced it to 15,360 bytes
- Delphi 7 produced a file of 15,872 bytes
- Delphi 2005 made a slightly bigger file at 16,384 bytes
- Both Delphi 2006 and Delphi 2007 make it grow to 19,456 bytes
- In Delphi 2009 the program size was 20,992 bytes.
- In Delphi 2010, it grows to 29,184 bytes.

The size difference (due to the RTTI for the compiler types and those defined in the `System` and `Windows` units) is quite relevant for a do-nothing program.

Now let's move one step ahead, and I'll recompile another programs I used in past books. This is called MiniPack and demonstrates a complete application, with its own main form, displaying the size of its executable in the caption. The name comes from the fact I compile this program with runtime packages, obtaining the following executable sizes:

- 17,408 bytes in Delphi 7
- 16,384 bytes in Delphi 2005
- 15,872 bytes in Delphi 2006 and 2007
- 16,384 bytes in Delphi 2009
- 17,920 bytes in Delphi 2010

This is certainly quite good, however it tells us very little about the effect of distributing a full blown application, using multiple units and built without runtime packages.

As a further example in the ExeSizeTest folder I've taken an existing program I wrote in Delphi 2009 for showing a ClientDataSet with Unicode strings. By using the ClientDataSet component and a DBGrid control, the program pulls in a fair amount of the VCL and RTL, including the database portion.

The increase in size for such a program is very significant:

- In Delphi 2009 it compiles to 1,025,024 bytes (about 1MB)
- In Delphi 2010 it takes 1,552,384 bytes (a little over 1.5MB)

If you are distributing internal applications, it won't be a big deal, but if your program is distributed in large numbers over the Internet, possibly with frequent updates, this increase might affect you in a significant way.

What can be done to reduce it? Cutting off Extended RTTI will decrease the size of your compiled program, but won't affect the library files. In the specific example (which has a single form with limited code, so it is not very significant) adding the \$RTTI directive shown earlier will trim only 1Kb from the executable, down to 1,551,360.

Weak and Strong Types Linking

What else could you do to reduce the size of the program, other then resorting to using Run Time Packages? There is actually something you can do, even if its effect won't be big, it will be noticeable.

When evaluating the RTTI information available in the executable file, consider that what the compiler adds, the linker might remove. By default, classes and method not compiled in the program will not get the Extended RTTI (which would be quite useless), as they don't get the basic RTTI either. At the opposite, if you want all Extended RTTI to be included and working, you need to link in even classes and methods you don't explicitly refer to in your code.

There are two new compiler settings you can use to control the information being linked into the executable. The first, which is fully documented, is the `$WeakLinkRTTI` directive. By turning it on, for types not used in the program, both the type itself and its RTTI information will be removed from the final executable. The previous program, even without the `$RTTI` directive, is reduced down to 1,414,144.

At the opposite, you can force the including of all type and their Extended RTTI using the undocumented `$StrongLinkTypes` directive. The effect on the program is dramatic, with almost a two fold increase to the already large program size, up to 2,807,296.

The following list summarizes the effect of this linker options on this program:

- Delphi 2009: 1,025,024 bytes
- Delphi 2010 (default settings) 1,552,384 bytes
- Delphi 2010 (with `$WeakLinkRTTI`) 1,414,144 bytes
- Delphi 2010 (with `$StrongLinkTypes`) 2,807,296 bytes

The Rtti Unit

If the generation of extended RTTI for all types is the first pillar for the new RTTI of Delphi 2010, the second pillar is the ability to navigate this information in a much easier and higher level way, thanks to the new Rtti unit. The third pillar, as we'll see, is the support for custom attributes. But let me proceed one step at a time.

Traditionally, Delphi applications could (and still can) use the functions of the `TypeInfo` unit to access the “published” run time type information. This unit defines several low-level data structures and functions (all based on pointers and records) with a couple of higher level routines to make things a little easier.

The new Rtti unit, instead, makes it very easy to work with the extended RTTI, providing a set of classes with proper methods and properties. To access the various objects, the entry point is the TRtti Context record structure, which has four methods to look for the available types:

```
function GetType (ATypeInfo: Pointer): TRttiType; overload;
function GetType (AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType (const AQualifiedName: string): TRttiType;
```

As you can see you can pass a class, a PTypeInfo pointer obtained from a type, a qualified name (the name of the type decorated with the unit name, as in “*System.TObject*”), or retrieve the entire list of types, defined as an array of Rtti types: TArray<TRttiType>¹⁹.

This last call is what I've done in the following listing, part of the TypesList example:

```
procedure TFormTypesList.btnTypesListClick(Sender: TObject);
var
  aContext: TRttiContext;
  theTypes: TArray<TRttiType>;
  aType: TRttiType;
begin
  ListBox1.Clear;
  theTypes := aContext.GetTypes;
  for aType in theTypes do
    if aType.IsInstance then
      ListBox1.Items.Add(aType.QualifiedName);
  ListBox1.Sorted := True;
end;
```

The GetTypes method returns the complete list of data types, but the program filters only the types representing classes (*instances* in the jargon used by the Rtti unit). There are about a dozen other classes representing types in the unit.

The individual objects in the types list are of classes which inherit from the TRttiType base class. Specifically, we can look for the TRttiInstanceType class type, as in the following modified snippet:

```
for aType in theTypes do
  if aType is TRttiInstanceType then
    ListBox1.Items.Add(aType.QualifiedName);
```

19 The notation used here is the instantiation of a generic class, in this case a generic dynamic array. Generics were introduced in Delphi 2009 and are used significantly in the Rtti unit, as we'll see in some examples. The topic is too complex to introduce in a footnote, so all I can do is refer you to the specific chapter on generics in my “Delphi 2009 Handbook”.

In the following list you can see the entire inheritance graph for the classes that derive from the abstract `TRttiObject` class and are defined in the `Rtti` unit:

```
TRttiObject (abstract)
  TRttiNamedObject
    TRttiType
      TRttiStructuredType (abstract)
        TRttiRecordType
        TRttiInstanceType
        TRttiInterfaceType
      TRttiOrdinalType
      TRttiEnumerationType
      TRttiInt64Type
      TRttiMethodType
      TRttiClassRefType
      TRttiSetType
      TRttiStringType
      TRttiAnsiStringType
      TRttiFloatType
      TRttiArrayType
      TRttiDynamicArrayType
      TRttiPointerType
      TRttiProcedureType
    TRttiMember
      TRttiField
      TRttiProperty
        TRttiInstanceProperty
      TRttiMethod
      TRttiParameter
      TRttiPackage
      TRttiManagedField
```

Each of these classes provides specific information about the given type. As an example, only a `TRttiInterfaceType` offers a way to access to the interface `GUID`. Notice, on the other hand, that there is no `Rtti` object to access indexed properties (like the `Strings[]` of a `TStringList`).

Rtti Objects Lifetime Management and the `TRttiContext` record

If you look at the source code of the `btnTypesListClick` method listed earlier, there is something that looks quite wrong. The `GetTypes` call returns an array of types, but the code doesn't free these internal objects. The reason is that the `TRttiContext` record structure becomes the effective owner for all of the `Rtti` objects that are being created. When the record is disposed (that is, when it goes out of scope), an internal interface is cleared invoking its own destructor that clears all of the `Rtti` objects that were created through it.

The TRtti Context record actually has a dual role. On one side it controls the lifetime of the Rtti objects (as I just explained), on the other hand it caches Rtti information that is quite expensive to recreate with a search. That's why you might want to keep reference to the TRtti Context record alive for an extended period, allowing you to keep accessing the Rtti objects it owns without having to recreate them (again, the expensive operation).

Internally the TRtti Context record uses a global pool of type TRtti Pool , which uses a critical section to make its access thread safe²⁰. So, to be more precise, the Rtti pool is shared among TRtti Context records, so the pooled Rtti objects are kept around while at least one TRtti Context record is in memory. To quote the comment in the unit:

```
{... working with RTTI objects without at least one context being
alive is an error. Keeping at least one context alive should keep
the Pool variable valid.}
```

In other words, you have to avoid caching and keeping Rtti objects around after you've released the Rtti context. This is an example that leads to an memory access violation (again part of the TypesList example):

```
function GetThisType (aClass: TClass): TRttiType;
var
    aContext: TRttiContext;
begin
    Result := aContext.GetType(aClass);
end;

procedure TFormTypesList.Button1Click(Sender: TObject);
var
    aType: TRttiType;
begin
    aType := GetThisType (TForm);
    ShowMessage (aType.QualifiedName);
end;
```

To summarize, the Rtti objects are managed by the context and you should not keep them around. The context in turn is a record, so it is disposed of automatically. You might see code that uses the TRtti Context in the following way:

```
context := TRttiContext.Create;
try
    // use the context
finally
    context.Free;
end;
```

²⁰ There are exceptions to the thread-safety of the Rtti pooling mechanism, described in some detail in the comments available in the Rtti unit itself.

The pseudo-constructor and pseudo-destructor set the internal interface, that manages the actual data structures used behind the scenes, to nil cleaning up the pooling mechanism. However, as this operation is automatic for a local type such as a record, this is not needed, unless somewhere you refer to the context record using a pointer. For more information about the internals of the TRtti Context record you can refer to the following blog post by Berry Kelly:

<http://blog.barrkel.com/2010/01/delphi-2010-rtti-contexts-how-they-work.html>

A Tree of Classes (and Class Information)

The most relevant types you might want to inspect at run time are certainly the so-called structured types, that is instances, interfaces, and records. Focusing on instances, we can refer to the relationship among classes, by following the BaseType information available for instance types.

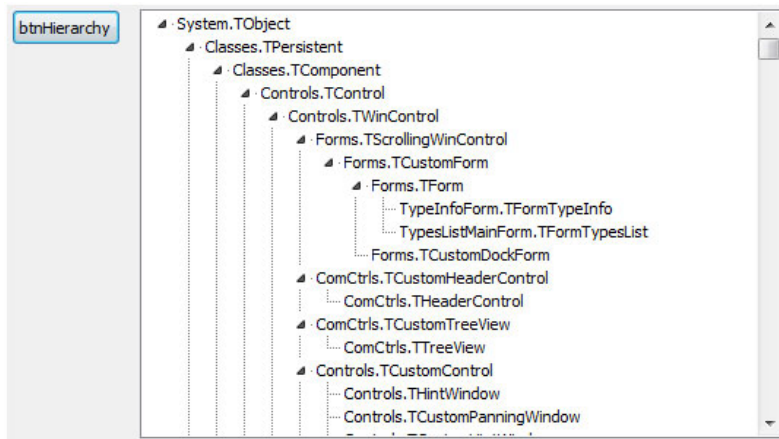
This is what I've done in the TypeList example, which can build a tree of the classes in a TreeView control, using the following recursive method:

```
function TFormTypesList.AddTypeToTree (
    atype: TRttiType): TTreeNode;
var
    ParentNode: TTreeNode;
begin
    // already there?
    Result := GetNodeFromTree (atype.name);

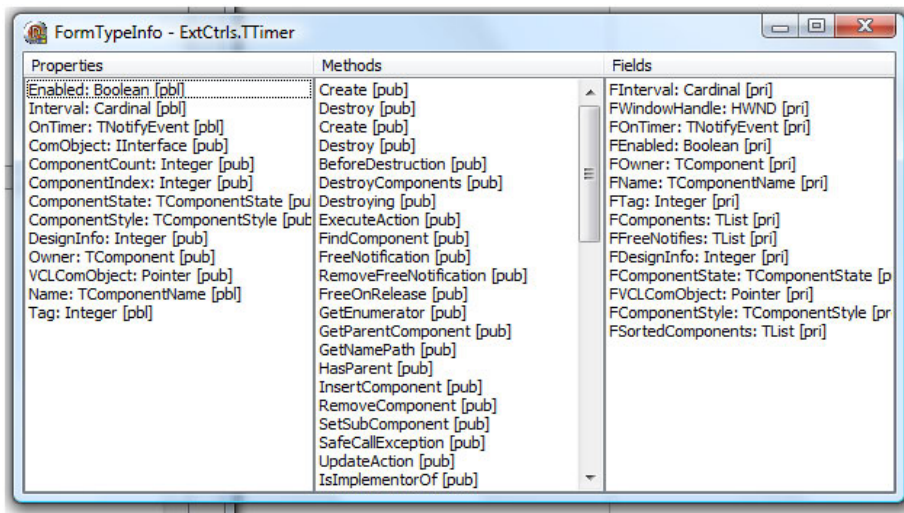
    if Result = nil then
    begin
        if atype.BaseType = nil then
            // add root node
            ParentNode := nil
        else
            // add the base class if not there
            ParentNode := AddTypeToTree (atype.BaseType);
            // now add the child class
            Result := TreeView1.Items.AddChild (ParentNode, atype.Name);
        end;
    end;
end;
```

At the beginning the method uses the GetNodeFromTree function to check if a type with the given name is already present in the tree, eventually returning the corresponding node. The next step is to look for or add the base class type to the tree, so that at the end the method can add the current node as a sub-node (or using nil as base node if it has no base class, as happens for TObject).

This `AddTypeToTree` method is invoked for each type representing an instance, with a loop similar to that we have already seen. Notice that we could have added some type information to each tree node, but in doing so we would have risked keeping the reference to these `Rtti` objects around after the context and the corresponding pooling mechanism would have deleted them. As an example, we can see the type of the demo form in this portion of the types tree:



Accessing types is certainly an interesting starting point, but what is relevant and specifically new is the ability to learn about further details of these types, including their members. As you click on one of the types (here the `TTimer` class) the program displays a list of properties, methods, and fields of the type:



The unit of this secondary form, which can probably be adapted and expanded to be used as a generic type browser in other applications, has a method called `ShowTypeInfo` that walks through each property, method, and field of the given type, adding them to three separate list boxes with the indication of their visibility (*pri* for private, *pro* for protected, *pub* for public, and *pbl* for published, as returned by the `VisibilityToken` function):

```

procedure ShowTypeInfo (aType: TRttiType);
var
  FormTypeInfo: TFormTypeInfo;
  aProperty: TRttiProperty;
  aMethod: TRttiMethod;
  aField: TRttiField;
begin
  FormTypeInfo := TFormTypeInfo.Create(nil);
  try
    FormTypeInfo.Caption := FormTypeInfo.Caption +
      ' - ' + aType.QualifiedName;
    for aProperty in aType.GetProperties do
      FormTypeInfo.ListProperties.Items.Add (aProperty.Name +
        ' : ' + aProperty.PropertyType.Name + ' ' +
        VisibilityToken (aProperty.Visibility));
    for aMethod in aType.GetMethods do
      FormTypeInfo.ListMethods.Items.Add (aMethod.Name + ' ' +
        VisibilityToken (aMethod.Visibility));
    for aField in aType.GetFields do
      FormTypeInfo.ListFields.Items.Add (aField.Name + ' : ' +
        aField.FieldType.Name + ' ' +
        VisibilityToken (aField.Visibility));
    FormTypeInfo.ShowModal;
  finally
    FormTypeInfo.Free;
  end;
end;

```

You could go ahead and extract further information from the types of these properties, get parameter lists of the methods and check the return type, and more. Here I don't want to build a complete RTTI browser but only give you a feeling of what can be achieved.

RTTI for Packages

Beside the methods you can use to access a type or the list of types, the record `TRttiContext` has another very interesting method, `GetPackages`, which returns a list of the run-time packages used by the current application. If you execute this method in an application compiled with no run time packages, all you get is the executable file itself. But if you execute it in an application com-

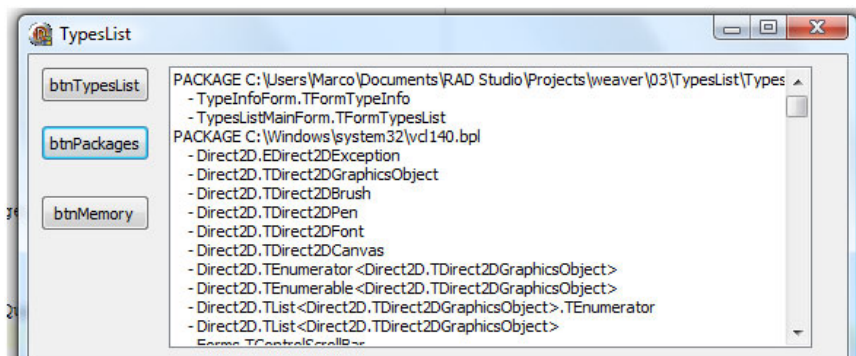
piled with run time packages, you'll get a list of those packages. From that point, you can delve into the types made available by each of the packages. Notice that in this case the types list is much larger, as RTL and VCL types not used by the application are not removed by the smart linker.

In the `TypesList` application, clicking on the `btnTypesList` button you'll get the list of instance types in the list box, and also the total number of types in the status bar. When the program is compiled statically, you'll get 300 instance types, while if you turn on run time packages that numbers becomes 800. Still, this is not the complete list of all classes available in the Delphi libraries, as you'll generally not include every available package, but only a limited set.

If you use run time packages, you can also retrieve the list of units for each of the packages (and the executable file), by using code like:

```
procedure TFormTypesList.btnPackagesClick(Sender: TObject);
var
  aContext: TRttiContext;
  aPackage: TRttiPackage;
  aType: TRttiType;
begin
  ListBox1.Clear;
  ListBox1.Sorted := False;
  for aPackage in aContext.GetPackages do
    begin
      ListBox1.Items.Add('PACKAGE ' + aPackage.Name);
      for aType in aPackage.GetTypes do
        if aType.IsInstance then
          begin
            ListBox1.Items.Add('    - ' + aType.QualifiedName);
          end;
        end;
      end;
    end;
end;
```

With this code you'll get the list of instance types for each package, starting with the executable, as you can see in the following image.



The TValue Structure

The new extended RTTI not only lets you browse the internal structure of a program but it also provides specific information, including property and field values. While the `TypeInfo` unit provided the `GetPropValue` function to access a generic property and retrieve a variant type with its value, the new `Rtti` unit uses a different structure for holding an untyped element, the `TValue` record.

This record can store almost any possible Delphi data type and does so by keeping track of the original data representation, by holding both a data and a format. What it can do is read and write data in the given format. What it cannot do is convert from one format to another. So even if a `TValue` has an `AsString` and an `AsInteger` method, you can use the former only if the data is representing is indeed a string, the second only if you originally assigned an integer to it. For example, in this case you can use the `AsInteger` method and if you call the `IsOrdinal` method it will return `True`:

```
var
  v1: TValue;
begin
  v1 := 100;
  if v1.IsOrdinal then
    Log (IntToStr (v1.AsInteger));
```

However, you cannot use the `AsString` method, which would raise an *invalid typecast* exception:

```
var
  v1: TValue;
begin
  v1 := 100;
  Log (v1.AsString);
```

If you need a string representation, though, you can use the `ToString` method, which has a large case statement trying to accommodate most data types:

```
var
  v1: TValue;
begin
  v1 := 100;
  Log (v1.ToString);
```

You can probably get a better understanding, by reading the words of Barry Kelly, Delphi R&D member who worked on RTTI for the Delphi 2010 compiler:

TValue is the type used to marshal values to and from RTTI-based calls to methods, and reads and writes of fields and properties.

It's somewhat like Variant but far more tuned to the Delphi type system; for example, instances can be stored directly, as well as sets, class references, etc. It's also more strictly typed, and doesn't do (for example) silent string to number conversions.

Now that you better understand its role, let's look at the actual capabilities of the `TValue` record. It has a set of higher level methods for assigning and extracting the actual values, plus a set of low-level pointer based ones. I'll concentrate on the first group.

For assigning values, `TValue` defines several `Implicit` operators, allowing you to perform a direct assignment as in the code snippets above:

```
class operator Implicit(const Value: string): TValue;
class operator Implicit(Value: Integer): TValue;
class operator Implicit(Value: Extended): TValue;
class operator Implicit(Value: Int64): TValue;
class operator Implicit(Value: TObject): TValue;
class operator Implicit(Value: TClass): TValue;
class operator Implicit(Value: Boolean): TValue;
```

What all these operators do is call the `From` generic class method:

```
class function From<T>(const Value: T): TValue; static;
```

When you call these class functions you need to specify the data type and also pass a value of that type, like the following code replacing the assignment of the value 100 of the previous code snippets:

```
v1 := TValue.From<Integer>(100);
```

This is a sort of universal technique for moving any data type into a `TValue`. Once the data has been assigned, you can use several methods to test its type:

```
property Kind: TTypeKind read GetTypeKind;
function IsObject: Boolean;
function IsClass: Boolean;
function IsOrdinal: Boolean;
function IsType<T>: Boolean; overload;
function IsArray: Boolean;
```

Notice that the generic `IsType` can be used for almost any data type.

There are corresponding method for extracting the data, but again you can use only the method compatible with the actual data stored in the `TValue`, as no conversion is taking place:

```
function AsObject: TObject;
function AsClass: TClass;
function AsOrdinal: Int64;
function AsType<T>: T;
function AsInteger: Integer;
```



```

function AsBoolean: Boolean;
function AsExtended: Extended;
function AsInt64: Int64;
function AsInterface: IInterface;
function AsString: string;
function AsVariant: Variant;
function AsCurrency: Currency;

```

Some of these methods double with a *Try* version that returns False, rather than raising an exception, in case of an incompatible data type. There are also some limited conversion methods, the most relevant of which are the generic Cast and the ToString function I've already used in the code:

```

function Cast<T>: TValue; overload;
function ToString: string;

```

Reading a Property with TValue

The importance of TValue lies in the fact that this is the structure used when accessing properties and field values using the extended RTTI and the Rtti unit in Delphi 2010. As an actual example of the use of TValue, we can use this record type to access both a published property and a private field of a TButton object, as in the following code (part of the RttiAccess demo):

```

procedure TForm39.btnReadValuesClick(Sender: TObject);
var
  context: TRttiContext;
  aType: TRttiType;
  aProperty: TRttiProperty;
  aValue: TValue;
  aField: TRttiField;
begin
  aType := context.GetType(TButton);
  aProperty := aType.GetProperty('Caption');
  aValue := aProperty.GetValue(Sender);
  ShowMessage(aValue.AsString);

  aField := aType.GetField('FDesignInfo');
  aValue := aField.GetValue(Sender);
  ShowMessage(IntToStr(aValue.AsInteger));
end;

```

Invoking Methods

Not only does the new extended RTTI lets you access values and fields, but it also provides a simplified way for calling methods. In this case you have to

define a `TValue` element for each parameter of the method. There is a global `Invoke` function which you can call for executing a method:

```
function Invoke(CodeAddress: Pointer; const Args: TArray<TValue>;
  CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

As a better alternative, there is a simplified `Invoke` overloaded method in the `TRttiMethod` class:

```
function Invoke(Instance: TObject;
  const Args: array of TValue): TValue; overload;
```

An example of invoking a method using this second simplified form is part of the `RttiAccess` demo and listed below:

```
procedure TForm39.btnInvokeClick(Sender: TObject);
var
  context: TRttiContext;
  aType: TRttiType;
  aMethod: TRttiMethod;
  theValues: array of TValue;
begin
  aType := context.GetType(TButton);
  aMethod := aType.GetMethod('FlipChildren');
  SetLength(theValues, 1);
  theValues[0] := True;
  aMethod.Invoke(self, theValues);
end;
```

Low-Level TValue

The `TValue` structure includes a few helpers that make it easy to assign or extract values of various data types to or from it. In some situations, however, it is nice to be able to copy the raw bytes of the `TValue` data into a specific data structure. This can be done using the `Make` and `GetRawData` methods, as in the following snippet that saves an integer value inside a (newly created) `TValue` record:

```
var
  aValue: TValue;
  intValue: Integer;
begin
  intValue := 100;
  TValue.Make(intValue, TypeInfo(Integer), aValue);
```

Of course, this is not needed as I could have used a higher level approach. At times, though, you won't have the alternative or, in some cases, the higher level approach would be more complicated than the low-level code.

As an example, consider the method (part of the `RttiAccess` application, like the code snippet above), used to read the value of a set property, the `Anchor`s property of a button. The `ToString` method of the `TValue` record lets you display a readable output, but if you need to read the actual numerical value (in this case a `Byte`) you can write the two lines at the end:

```
var
  context: TRttiContext;
  aType: TRttiType;
  aSetType: TRttiSetType;
  aProperty: TRttiProperty;
  aValue: TValue;
  b: Byte;
begin
  aType := context.GetType(TButton);
  aProperty := aType.GetProperty('Anchor');
  aSetType := aProperty.PropertyType.AsSet;
  Log('Type: ' + aSetType.ToString);

  aValue := aProperty.GetValue(Sender);
  Log('Anchor: ' + aValue.ToString);

  // extract numerical value
  aValue.ExtractRawData(@b);
  Log('Anchor: ' + IntToStr(b));
end;
```

The output of this method is:

```
Type: TAnchor
Anchor: [akLeft, akTop]
Anchor: 3
```

In most cases the higher level approach would work, but in specific circumstances it simply won't. Trying to get the numerical value of the set by using the `GetOrdinal` method of `TValue`, for example, will raise an exception because the set is not an ordinal value. If you want to skip similar exceptions and just read (or write) the data, you'll need to use the low-level calls.

Custom Attributes

The first part of this chapter gave you a good grasp of the extended RTTI generated by the Delphi 2010 compiler and of the RTTI access capabilities introduced by the new `Rtti` unit. In the second part of the chapter we can finally focus on one of the key reasons this entire architecture was introduced: the

possibility to define custom attributes and extend the compiler-generated RTTI in specific ways. We'll look at this technology from a rather abstract perspective, and later focus on the reasons this is an important step forward for Delphi, by looking at practical examples.

What is an Attribute?

An attribute (in Delphi or .NET terms) or an annotation (in Java Jargon) is a comment or indication that you can add to your source code, applying it to a type, a field, a method, or a property) and the compiler will embed in the program. This is generally indicated with square brackets as in:

```
type
  [MyAttribute]
  TMyClass = class
    ...
```

By reading this information at design time in a development tool or at run time in the final application, a program can change its behavior depending on the values it finds.

Generally attributes are not used to change the actual core capabilities of a class of objects, but rather to let these classes specify further mechanisms they can participate in. Declaring a class as *serializable* doesn't affect its code in any way, but lets the serialization code know that it can operate on that class and how (in case you provide further information along with the attribute, or further attributes marking the class fields or properties).

This is exactly how the existing and limited RTTI was used inside Delphi. Properties marked as published could show up in the object inspector, be streamed to a DFM file, and be accessed at run time. Attributes open up this mechanism to become much more flexible and powerful. They are also much more complex to use, and easy to misuse, as are any powerful language features. I mean, don't throw away all the good things you know about Object Oriented Programming to embrace this new model, but complement one with the other.

As an example, an employee class will still be represented in a hierarchy as a derived class from a person class; an employee object will still have an ID for his or her badge; but you can “mark” or “annotate” the employee class as class that can be mapped to a database table or displayed by a specific runtime form.

So we have inheritance (is-a), ownership (has-a), and annotations (marked-as) as three separate mechanism you can use when designing an application.

After you've seen the compiler features supporting custom attributes in Delphi 2010 and looked at some practical examples, the abstract idea I just mentioned should become more understandable, or at least that's my hope!

Attribute Classes and Attribute Declarations

How do you define a new attribute class (or attribute category)? You have to inherit from the new `TCustomAttribute` class available in the `System` unit:

```
type
  SimpleAttribute = class(TCustomAttribute)
  end;
```

The class name you give to the attribute class will become the symbol to use in the source code, with the optional exclusion of the *Attribute* postfix. So if you name your class `SimpleAttribute` you'll be able to use in the code an attribute called `Simple` or `SimpleAttribute`. Anyway this is the reason the classic initial `T` for Delphi classes is generally not used in case of attributes.

Now that we have defined a custom attribute, we can apply it to most of the symbols of our program: types, methods, properties, fields, and parameters. The syntax used for applying attributes is the attribute name within square brackets:

```
type
  [Simple]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
```

In this case I've applied the `Simple` attribute to the class as a whole and to a method. Beside a name, attribute can support one or more parameters. The parameters passed to an attribute must match those indicated in the constructor of the attribute class, if any.

```
type
  ValueAttribute = class(TCustomAttribute)
  private
    FValue: Integer;
  public
    constructor Create(N: Integer);
    property Value: Integer read FValue;
  end;
```

This is how you can apply this attribute with one parameter:

```
type
  [Value(22)]
  TMyClass = class(TObject)
  public
    [Value(0)]
    procedure Two;
```

The attribute values, passed to its constructor, must be constant expressions, as they are resolved at compile time. That's why you are limited to just a few data types: ordinal values, strings, sets, and class references. On the positive side, you can have multiple overloaded constructors with different parameters.

Notice you can apply multiple attributes to the same symbol, as I've done in the `RttiAttrib` example, which summarizes the code snippets of this section:

```
type
  [Simple][Value(22)]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
    [Value(0)]
    procedure Two;
  end;
```

What if you try to use an attribute that is not defined (maybe because of a missing uses statement)? Unluckily you get a very misleading warning message:

```
[DCC Warning] RttiAttribMainForm.pas(44): W1025
  Unsupported language feature: 'custom attribute'
```

The fact this is a warning implies the attribute will be ignored, so you have to watch out for those warnings or even better treat the “unsupported language feature” warning like an error (something you can do in the Hints and Warnings page of the Project Options dialog box):

```
[DCC Error] RttiAttribMainForm.pas(38):
  E1025 Unsupported language feature: 'custom attribute'
```

Finally, compared to other implementations of the same concept, there is currently no way to limit the scope of attributes, like declaring that an attribute can be applied to a type but not to a method. What is available in the editor, instead, is full support for attributes in the rename refactoring²¹. Not only you can change the name of the attribute class, but the system will pick up when the attribute is used both in its full name and without the final “*attribute*” portion.

21 Attributes refactoring was first mentioned by Malcolm Groves on his blog at <http://www.malcolmgroves.com/blog/?p=554>

Browsing Attributes

Now this code would seem totally useless if there wasn't a way to discover which attributes are defined, and possibly inject a different behavior to an object because of these attributes. Let me start focusing on the first part. The classes of the Rtti unit let you figure out which symbols have associated attributes. This is code, extracted from the RttiAttrib example shows the list of the attributes for the current class:

```
procedure TMyClass.One;
var
  context: TRttiContext;
  attributes: TArray<TCustomAttribute>;
  attrib: TCustomAttribute;
begin
  attributes := context.GetType(ClassType).GetAttributes;
  for attrib in attributes do
    Form39.Log(attrib.ClassName);
```

Running this code will print out:

```
SimpleAttribute
ValueAttribute
```

You can extend it by adding the following code to the for-in loop code to extract the specific value of the given attributes type:

```
if attrib is ValueAttribute then
  Form39.Log(' - ' + IntToStr
    (ValueAttribute(attrib).Value));
```

What about fetching the methods with a given attribute, or with any attribute? You cannot filter the methods up front, but have to go through each of them, check their attributes, and see if it is relevant for you. To help in this process, I've written a function that checks if a method supports a given attribute:

```
type
  TCustomAttributeClass = class of TCustomAttribute;

function HasAttribute (aMethod: TRttiMethod;
  attribClass: TCustomAttributeClass): Boolean;
var
  attributes: TArray<TCustomAttribute>;
  attrib: TCustomAttribute;
begin
  Result := False;
  attributes := aMethod.GetAttributes;
  for attrib in attributes do
    if attrib.InheritsFrom (attribClass) then
      Exit (True);
end;
```

The `HasAttribute` function is called by the `RttiAttr` program while checking for a given attribute or any of them:

```
var
  context: TRttiContext;
  aType: TRttiType;
  aMethod: TRttiMethod;
begin
  aType := context.GetType(TMyClass);

  for aMethod in aType.GetMethods do
    if HasAttribute(aMethod, SimpleAttribute) then
      Log(aMethod.name);

  for aMethod in aType.GetMethods do
    if HasAttribute(aMethod, TCustomAttribute) then
      Log(aMethod.name);
```

The effect is to list methods marked with the given attributes, as described by further `Log` calls I've omitted from the listing above:

```
Methods marked with [Simple] attribute
One

Methods marked with any attribute
One
Two
```

Rather than simply describing attributes, what you generally do is add some independent behavior determined by the attributes of a class, rather than its actual code. As an example, I can inject a specific behavior in the previous code: The goal could be calling all methods of a class marked with a given attribute, considering them as parameterless methods:

```
procedure TForm39.btnInvokeClick(Sender: TObject);
var
  context: TRttiContext;
  aType: TRttiType;
  aMethod: TRttiMethod;
  aTarget: TMyClass;
  zeroParams: array of TValue;
begin
  aTarget := TMyClass.Create;
  try
    aType := context.GetType(aTarget.ClassType);
    for aMethod in aType.GetMethods do
      if HasAttribute(aMethod, SimpleAttribute) then
        aMethod.Invoke(aTarget, zeroParams);
  finally
    aTarget.Free;
  end;
end;
```


What this code snippet does is create an object, grab its type, check for a given attribute, and invoke each method that has the `Simple` attribute. Rather than inheriting from a base class, implementing an interface, or writing specific code to perform the request, all we have to do to get the new behavior is mark one of more methods with a given attribute. Not that this example makes the use of attributes extremely obvious: for some common patterns in using attributes and some actual case studies you can refer to the final part of this chapter.

RTTI Case Studies

Now that I've covered the foundations of RTTI and the use of attributes it is worth looking into some real world situations in which using these technique will prove useful. There are many scenarios in which a more flexible RTTI and the ability to customize it through attributes is relevant, but I have no room for a long list of situations. What I'll do instead is guide you in the step-by-step development of two simple but significant examples.

The first demo program will showcase the use of attributes to identify specific information within a class. In particular, we want to be able to inspect an object of a class that declares to be part of the an architecture and have a description and a unique ID referring to the object itself. This might come handy in several situations, like describing objects stored in a collection (either a generic or traditional one).

The second demo will be an example of streaming, specifically streaming a class to an XML file. I'll start from the classic approach of using the published RTTI, move to the new extended RTTI, and finally show how you can use attributes to customize the code and make it more flexible.

Attributes for ID and Description

If you want to have a couple of methods shared among many objects, the most classic approach was to define a base class with virtual methods and inherit the various objects from the base class, overriding the virtual methods. This is nice, but poses a lot of restrictions in terms of the classes which can participate in the architecture, as you have a fixed base class.

A standard technique to overcome this situation is to use an interface rather than a common base class. Multiple classes implementing the interface (but with no common ancestor class) can provide an implementation of the interface methods, which act very similarly to virtual methods.

A totally different style (with both advantages and disadvantages) is the use of attributes to mark participating classes and given methods (or properties). This opens up more flexibility, doesn't involve interfaces, but is based on a comparatively slow and error-prone run-time information look up, rather than a compile-time resolution. This means I'm not advocating this coding style over interfaces as a better approach, only as one that might be worth evaluating and interesting to use in some circumstances.

The Description Attribute Class

For this demo, I've defined an attribute with a setting indicating the element is it being applied to. I could have used three different attributes, but prefer to avoid polluting the attributes name space. This is the attribute class definition:

```
type
  TDescriptionAttrKind = (dakClass, dakDescription, dakId);

  DescriptionAttribute = class (TCustomAttribute)
  private
    fDak: TDescriptionAttrKind;
  public
    constructor Create (aDak: TDescriptionAttrKind = dakClass);
    property Kind: TDescriptionAttrKind read fDak;
  end;
```

Notice the use of the constructor with a default value for its only parameter, to let you use the attribute with no parameters.

The Sample Classes

Next I wrote two sample classes that use the attribute. Each class is marked with the attribute and has two methods marked with the same attribute customized with the different *kinds*. The first (TPerson) has the description mapped to the GetName function and uses its TObject.GetHashCode method to provide a (temporary) ID, re-declaring the method to apply the attribute to it (the method code is simply a call to the inherited version):

```
type
  [Description]
  TPerson = class
```

```

private
    FBirthDate: TDate;
    FName: string;
    FCountry: string;
    procedure SetBirthDate(const Value: TDate);
    procedure SetCountry(const Value: string);
    procedure SetName(const Value: string);
public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetStringCode: Integer;
published
    property Name: string read GetName write SetName;
    property BirthDate: TDate
        read FBirthDate write SetBirthDate;
    property Country: string read FCountry write SetCountry;
end;

```

The second class (TCompany) is even simpler as it has its own values for the ID and the description:

```

type
    [Description]
    TCompany = class
    private
        FName: string;
        FCountry: string;
        FID: string;
        procedure SetName(const Value: string);
        procedure SetID(const Value: string);
    public
        [Description (dakDescription)]
        function GetName: string;
        [Description (dakID)]
        function GetID: string;
    published
        property Name: string read GetName write SetName;
        property Country: string read FCountry write FCountry;
        property ID: string read FID write SetID;
    end;

```

Even if there are similarities among the two classes they are totally unrelated in terms of hierarchy, common interface, or anything like that. What they share is the use of the same attribute.

The Sample Project and Attributes Navigation

The shared use of the attribute is used to display information about objects added to a list, declared in the main form of the program as:

```

fObjectsList: TObjectList<TObject>;

```

This list is created and initialized as the program starts:

```

procedure TFormDescrAttr. FormCreate(Sender: TObject);
var
    aPerson: TPerson;
    aCompany: TCompany;
begin
    fObjectsList := TObjectList<TObject>.Create;

    // add a person
    aPerson := TPerson.Create;
    aPerson.Name := 'Wiley';
    aPerson.Country := 'Desert';
    aPerson.BirthDate := Date - 1000;
    fObjectsList.Add(aPerson);

    // add a company
    aCompany := TCompany.Create;
    aCompany.Name := 'ACME Inc.';
    aCompany.ID := IntToStr(GetTickCount);
    aCompany.Country := 'Worldwide';
    fObjectsList.Add(aCompany);

    // add an unrelated object
    fObjectsList.Add(TStringList.Create);

```

To display information about the objects (namely the ID and the description, if available) the program uses attributes discovery via RTTI. First, it uses a helper function to determine if the class is marked with the specific attribute²²:

```

function TypeHasDescription (aType: TRttiType): Boolean;
var
    attrib: TCustomAttribute;
begin
    for attrib in aType.GetAttributes do
        begin
            if (attrib is DescriptionAttribute) then
                Exit (True);
        end;
    Result := False;
end;

```

If this is the case, the program proceeds by getting each attribute of each methods, with a nested loop, and checking if this is the attribute we are looking for:

```

if TypeHasDescription (aType) then
    begin
        for aMethod in aType.GetMethods do
            for attrib in aMethod.GetAttributes do
                if attrib is DescriptionAttribute then
                    ...
    end;

```

²² In this case you need to check for the full class name, `DescriptionAttribute`, and not only “Description”, which is the symbol you can use when applying the attribute.

At the core of the loop, the methods marked with attributes are invoked to read the results in two temporary strings (later added to the user interface):

```

if attrib is DescriptionAttribute then
  case DescriptionAttribute(attrib).Kind of
    dakClass: ; // ignore
    dakDescription:
      strDescr := aMethod.Invoke(anObject, []).ToString;
    dakId:
      strID := aMethod.Invoke(anObject, []).ToString;

```

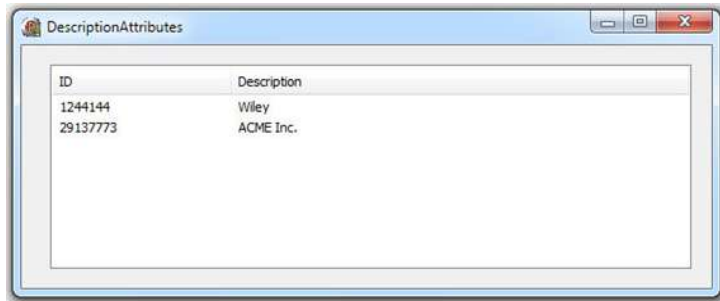
What the program fails to do is to check if an attribute is duplicated (that is, if there are multiple methods marked with the same attribute, a situation in which you might want to raise an exception). Summing up all of the snippets of the previous page, this is the complete code of the UpdateList method:

```

procedure TFormDescrAttr.UpdateList;
var
  anObject: TObject;
  context: TRttiContext;
  aType: TRttiType;
  attrib: TCustomAttribute;
  aMethod: TRttiMethod;
  strDescr, strID: string;
begin
  for anObject in fObjectsList do
    begin
      aType := context.GetType(anObject.ClassInfo);
      if TypeHasDescription(aType) then
        begin
          for aMethod in aType.GetMethods do
            for attrib in aMethod.GetAttributes do
              if attrib is DescriptionAttribute then
                case DescriptionAttribute(attrib).Kind of
                  dakClass: ; // ignore
                  dakDescription:
                    // should check if duplicate attribute
                    strDescr := aMethod.Invoke(
                      anObject, []).ToString;
                  dakId:
                    strID := aMethod.Invoke(
                      anObject, []).ToString;
                end;
              // done looking for attributes
              // should check if we found anything
              with ListView1.Items.Add do
                begin
                  Caption := strID;
                  SubItems.Add(strDescr);
                end;
            end;
          end;
        end;
      // else ignore the object, could raise an exception
    end;

```

If this program produces rather uninteresting output, shown below, the way it is done is relevant, as I've marked some classes and two methods of those classes with an attribute, and have been able to process these classes with an external algorithm. In other words, the classes themselves need no specific base class, no interface implementation nor any internal code to be part of the architecture, but only need to *declare* they want to get involved. The full responsibility for managing the classes is in some external code.



XML Streaming

One interesting and very ample case for RTTI is creating an “*external*” image of an object, for saving its status to a file or sending it over the wire to another application. Traditionally, the Delphi approach to this problem has been streaming the published properties of an object, the same approach used when creating DFM files. Now the RTTI lets you save the actual data of the object, its fields, rather than the external interface. This is more powerful, although it can lead to extra complexity, for example in the management of the data of internal objects. Again, the demo acts as a simple showcase of the technique and doesn't delve into all of its implications.

This examples comes in three versions, compiled in a single project for simplicity. The first is the traditional Delphi approach based on published properties, the second uses the extended RTTI and fields, the third uses attributes to customize the data mapping.

The Trivial XML Writer Class

To help with the generation of the XML, I've based the XmlPersist demo on an extended version of a TTrivialXmlWriter class I originally wrote in my Delphi 2009 Handbook to demonstrate the use of the TTextWriter class.

Here I don't want to cover it again. Suffice to say that the class can keep track of the XML nodes it opens, thanks to a stack of strings, and close the XML nodes in a LIFO (Last In, First Out) order.²³

To the original class I've added some limited formatting code and three methods for saving an object, based on the three different approaches I'm going to explore in this section. This is the complete class declaration:

```
type
  TTrivialXmlWriter = class
  private
    fWriter: TTextWriter;
    fNodes: TStack<string>;
    fOwnsTextWriter: Boolean;
  public
    constructor Create (aWriter: TTextWriter); overload;
    constructor Create (aStream: TStream); overload;
    destructor Destroy; override;
    procedure WriteStartElement (const sName: string);
    procedure WriteEndElement (fIndent: Boolean = False);
    procedure WriteString (const sValue: string);
    procedure WriteObjectPublished (AnObj: TObject);
    procedure WriteObjectRtti (AnObj: TObject);
    procedure WriteObjectAttrib (AnObj: TObject);
    function Indentation: string;
  end;
```

To get an idea of the code, this is the `WriteStartElement` method, which uses the `Indentation` function for leaving twice as much spaces as the current number of nodes on the internal stack:

```
procedure TTrivialXmlWriter.WriteStartElement(
  const sName: string);
begin
  fWriter.Write (Indentation + '<' + sName + '>');
  fNodes.Push (sname);
end;
```

You'll find the complete code of the class in the project source code.

Classic RTTI-Based Streaming

After this introduction covering the support class, let me start from the very beginning, that is saving an object in an XML-based format using the classic RTTI for published properties.

²³ The source code of the `TTrivialXmlWriter` class of Delphi 2009 Handbook can be found at <http://www.marcocantu.com/code/dh2009/ReaderWriter.htm>

The code of the `WriteObjectPublished` method is quite complex and requires a bit of explanation. It is based on the `TypeInfo` unit and uses the low-level version of the old RTTI to be able to get the list of published properties for a given object (the `AnObj` parameter), with code like:

```
nProps := GetTypeData(AnObj.ClassInfo).PropCount;
GetMem(PropList, nProps * SizeOf(Pointer));
GetPropInfos(AnObj.ClassInfo, PropList);
for i := 0 to nProps - 1 do
  ...
```

What this does is ask for the number of properties, allocate a data structure of the proper size, and fill the data structure with information about the published properties. In case you are wondering could you write this low-level code? Well you've just found a very good reason why the new RTTI was introduced. For each property, the program extracts the value of properties of numeric and string types, while it extracts any sub-object and acts recursively on it:

```
strPropName := UTF8ToString (PropList[i].Name);
case PropList[i].PropType^.Kind of
  tkInteger, tkEnumeration, tkString, tkUString, ...:
    begin
      WriteStartElement (strPropName);
      WriteString (GetPropValue(AnObj, strPropName));
      WriteEndElement;
    end;
  tkClass:
    begin
      internalObject := GetObjectProp(AnObj, strPropName);
      // recurse in subclass
      WriteStartElement (strPropName);
      WriteObjectPublished (internalObject as TPersistent);
      WriteEndElement (True);
    end;
end;
```

There is some extra complexity, but for the sake of the example and to give you an idea of the traditional approach, that should be enough.

To demonstrate the effect of the program I've written two classes (`TCompany` and `TPerson`) adapted from the previous example. This time, however, the company can have a person assigned to an extra property, called `Boss`. In the real world this would be more complex, but for this example it is a reasonable assumption. These are the published properties of the two classes:

```
type
  TPerson = class (TPersistent)
    ...
  published
    property Name: string read FName write FName;
```



```

    property Country: string read FCountry write FCountry;
end;

TCompany = class (TPersistent)
..
published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write FID;
    property Boss: TPerson read FPerson write FPerson;
end;

```

The main form of the program has a button used to create and connect two objects of these two classes and saving them to an XML stream, which is later displayed. The streaming section has the following code:

```

ss := TStringStream.Create;
xmlWri := TTrivialXmlWriter.Create(ss);
xmlWri.WriteStartElement('company');
xmlWri.WriteObjectPublished(aCompany);
xmlWri.WriteEndElement;

```

The result is an XML file like:

```

<company>
  <Name>ACME Inc.</Name>
  <Country>Worldwide</Country>
  <ID>29088851</ID>
  <Boss>
    <Name>Wiley</Name>
    <Country>Desert</Country>
  </Boss>
</company>

```

Streaming Fields With the New RTTI

Now that Delphi 2010 provides us with a much higher-level RTTI, I can convert the program to use this new RTTI for accessing the published properties. What I'm going to do, instead, is to use it for saving the internal representation of the object, that is, its private data fields. Not only am I doing something more hard-core, but I'm doing it with much higher-level code. The complete code of the `WriteObjectRtti` method is the following:

```

procedure TTrivialXmlWriter.WriteObjectRtti (AnObj: TObject);
var
  aContext: TRttiContext;
  aType: TRttiType;
  aField: TRttiField;
begin
  aType := aContext.GetType (AnObj.ClassType);
  for aField in aType.GetFields do
    begin

```

```

if aField.FieldType.IsInstance then
begin
  WriteStartElement (aField.Name);
  WriteObjectRtti (aField.GetValue(anObj).AsObject);
  WriteEndElement (True);
end
else
begin
  WriteStartElement (aField.Name);
  WriteString (aField.GetValue(anObj).ToString);
  WriteEndElement;
end;
end;
end;

```

The resulting XML is somewhat similar, but somehow less clean as field names are generally less readable than property names:

```

<company>
  <FName>ACME Inc. </FName>
  <FCountry>Worldwide</FCountry>
  <FID>29470148</FID>
  <FPerson>
    <FName>Wiley</FName>
    <FCountry>Desert</FCountry>
  </FPerson>
</company>

```

Another big difference, though, is that in this case the classes didn't need to inherit from the `TPersistent` class or be compiled with any special option.

Using Attributes to Customize Streaming

Beside the problem with the tag names, there is another issue I haven't mentioned. Using XML tag names which are actually compiled symbols is far from a good idea. Also, in the code there is no way to exclude some properties²⁴ or fields from XML-base streaming. These are issues we can address using attributes, although the drawback will be having to use them quite heavily in the declaration of our classes, a coding style I don't like much. For the new version of the code, I've defined an attribute constructor with an optional parameter:

```

type
  xmlAttribute = class (TCustomAttribute)
  private
    fTag: string;
  public

```

24 Delphi properties streaming can be controlled using the `stored` directive, which can be read using the `TypeInfo` unit. Still, this solution is far from simple and clean, even if the DFM streaming mechanism uses it effectively.

98 - Chapter 3: Extended RTTI and Attributes

```
    constructor Create (strTag: string = '');  
    property TagName: string read fTag;  
end;
```

The attributes-based streaming code is a variation of the last version based on the extended RTTI. The only difference is that now the program calls the `CheckXmlAttribute` helper function to verify if the field has the `xml` attribute and the (optional) tag name decoration:

```
procedure TTrivialXmlWriter.WriteObjectAttrib(AnObj: TObject);  
var  
    aContext: TRttiContext;  
    aType: TRttiType;  
    aField: TRttiField;  
    strTagName: string;  
begin  
    aType := aContext.GetType(AnObj.ClassType);  
    for aField in aType.GetFields do  
        begin  
            if CheckXmlAttribute(aField, strTagName) then  
                begin  
                    if aField.FieldType.IsInstance then  
                        begin  
                            WriteStartElement(strTagName);  
                            WriteObjectAttrib(aField.GetValue(AnObj).AsObject);  
                            WriteEndElement(True);  
                        end  
                    else  
                        begin  
                            WriteStartElement(strTagName);  
                            WriteString(aField.GetValue(AnObj).ToString);  
                            WriteEndElement;  
                        end  
                    end;  
                end;  
            end;  
        end;  
end;
```

The most relevant code is in the `CheckXmlAttribute` helper function:

```
function CheckXmlAttribute(aField: TRttiField;  
    var strTag: string): Boolean;  
var  
    attrib: TCustomAttribute;  
begin  
    Result := False;  
    for attrib in aField.GetAttributes do  
        if attrib is XmlAttribute then  
            begin  
                strTag := xmlAttribute(attrib).TagName;  
                if strTag = '' then // default value  
                    strTag := aField.Name;  
                Exit(True);  
            end;  
        end;  
    end;  
end;
```

Fields without the XML attribute are ignored and the tag used in the XML output is customizable. To demonstrate this, the program has the following classes (this time I've omitted the published properties from the listing, as they are not relevant):

```
type
  TAttrPerson = class
  private
    [xml ('Name')]
    FName: string;
    [xml]
    FCountry: string;
    ...

  TAttrCompany = class
  private
    [xml ('CompanyName')]
    FName: string;
    [xml ('Country')]
    FCountry: string;
    FID: string; // omitted
    [xml ('TheBoss')]
    FPerson: TAttrPerson;
    ...
```

With these declarations, the XML output will look like the following (notice the tag name, the fact the ID is omitted, and the (bad looking) default name for the FCountry field):

```
<company>
  <CompanyName>ACME Inc. </CompanyName>
  <Country>Worldwide</Country>
  <TheBoss>
    <Name>Wiley</Name>
    <FCountry>Desert</FCountry>
  </TheBoss>
</company>
```

The difference here is we can be very flexible about which fields to include and how to name them in the XML, something the previous versions didn't allow.

Even if this is just a very skeletal implementation, I think that giving you the opportunity to see the final version being created step by step starting with the classic RTTI has given you a good feeling of the differences among the various techniques. What is important to keep in mind, in fact, is that it is not a given that using attributes will be always the best solution! On the other hand, it should be clear that RTTI and attributes add a lot of power and flexibility in any scenario in which you need to inspect the structure of an unknown object at run time.

What's Next

For its long term viability, Delphi must provide new language features that can be used as a renewed foundation for the development of modern frameworks. The growth of framework-based code is quite significant both in the Java world and in the .NET environment, and missing the language features that are at the foundations of these frameworks was quite a negative point for Delphi. Now with generics and anonymous methods introduced in Delphi 2009 and attributes added to Delphi 2010 the gap is reducing significantly. So I hope that the next few years will see the start of new Delphi frameworks or the port to Delphi of existing open source frameworks used by other environments. Both Embarcadero Technologies as a company and the Delphi community at large should push for this goal.

That's why the current chapter is probably the most relevant (and also one of the longest) of the entire book. In it I've fully delved into the new Extended RTTI of Delphi 2010 and its Attribute support. There are some other new compiler features that are worth exploring, which is what I'll do in the first part of the next chapter.

The second part of Chapter 4 focuses on changes to the Run Time Library (RTL, for short). In Chapter 5 I'll start exploring new features of the Visual Component Library (VCL, for short).

Chapter 4: More On The Compiler And The RTL

If Extended RTTI and attribute support are the most significant new features of the Delphi 2010 compiler, there are three other important enhancements: the ability to access the object behind an interface reference, delayed loading of DLL functions, and class constructors. I'll look at these features in the first part of this chapter. The second part of the chapter will focus on the RTL and the new IOUtils unit, which defines classes for files and folders.

New Compiler Features

While it is possible for third-parties to provide custom components or extend the IDE, the compiler is the core capability that only the R&D Team can extend. That's why each change at the compiler level is worth special consideration.

Version

First of all, in case you need to have some code specifically tied to Delphi 2010, consider that while the IDE version number is 14, the compiler version number (dating back to the original Turbo Pascal) is 21. The corresponding compiler define is `VER210`, so you can have code that compiles only for this version with:

```
if defined(VER210)
```

Extracting Objects from Interface References

It was the case for many versions of Delphi, that when you assign an object to an interface variable, there was no way to access the original object. At times, Delphi developers would add a `GetObject` method to their interfaces to perform the operation, but that is quite an odd design.

For the first time since interfaces were introduced, Delphi 2010 adds support for casting interface references back to the original object to which they have been assigned. There are three separate operations you can use:

- You can write an `is` test to verify that an object of the given type can indeed be extracted from the interface reference:

```
if IntfVar is TMyObject.
```
- You can write an `as` cast to perform the type cast, raising an exception in case of an error:

```
if IntfVar as TMyObject.
```
- You can write a hard type cast to perform the same conversion, returning a `nil` pointer in case of an error:

```
TMyObject(IntfVar)
```

In every case, the type cast operation works only if the interface was originally obtained from a Delphi object, and not from a COM server. Note also that you can not only cast to the exact class of the original object, but also to one of its base classes (following standard class compatibility rules for derived classes).

As an example, consider having the following simple interface and implementation class (part of the `ObjFromIntf` project):

```
type
  ITestIntf = interface (IInterface)
    ['{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}']
    procedure DoSomething;
  end;
```

```

TTestImpl = class (TInterfacedObject, ITestIntf)
public
  procedure DoSomething;
  procedure DoSomethingElse; // not in interface
  destructor Destroy; override;
end;

```

With these definitions you can now define an interface variable, assign an object to it, and use it also to invoke the method not in the interface, with the new cast:

```

var
  intf: ITestIntf;
begin
  intf := TTestImpl.Create;
  intf.DoSomething;
  (intf as TTestImpl).DoSomethingElse;

```

You can also write the code in the following way, using an `is` test and a direct cast, and you can always cast to a base class of the actual class of the object:

```

var
  intf: ITestIntf;
  original: TObject;
begin
  intf := TTestImpl.Create;
  intf.DoSomething;
  if intf is TObject then
    original := TObject(intf);
    (original as TTestImpl).DoSomethingElse;

```

Considering that a direct cast returns `nil` if not successful, you could also write the code as follows (without the previous `is` test):

```

original := TObject(intf);
if Assigned(original) then
  (original as TTestImpl).DoSomethingElse;

```

Notice that assigning the object extracted from the interface to a variable exposes you to reference counting issues: when the interface is set to `nil` or goes out of scope, the object is actually deleted and the variable referring to it will become invalid. You'll find the code highlighting the problem in the `btnRefCountIssueClick` event handler of the example.

Technically, the three operations (`as` cast, direct conversion, `is` test) are implemented by three new global routines of the `System` unit:

```

function _IntfAsClass(const Intf: IInterface;
  Parent: TClass): TObject;
function _SafeIntfAsClass(const Intf: IInterface;
  Parent: TClass): TObject;
function _IntfIsClass(const Intf: IInterface;
  Parent: TClass): Boolean;

```

Class Constructors (and Destructors)

Class constructors are a new feature that has been borrowed from the .NET environment and were already available in Delphi for .NET. A class constructor has nothing to do with a standard constructor (or instance constructor): It is merely code used to initialize the class itself once (generally class data or other global settings) before the class is used.

In other words, a class constructor is an alternative to the unit initialization code. In case both exist (in a unit), the class constructor will be executed first. At the opposite, you can define a class destructor that will be executed after the finalization code.

A significant difference, however, is that while the unit initialization code is invariably executed if the unit is compiled in the program, the class constructor and destructor are linked only if the class is actually used. This means that the use of class constructor is much more linker friendly than the use of initialization code. With class constructors and destructors, if the type is not linked the initialization code is not part of the program and not executed; in the traditional case the opposite is true, the initialization code might even cause the linker to bring in some of the class code, even if it is never actually used²⁵.

In terms of code, you can write the following (taken from the ClassCtor demo):

```
type
  TTestClass = class
  public
    class var
      StartTime: TDateTime;
      EndTime: TDateTime;
    public
      class constructor Create;
      class destructor Destroy;
  end;
```

The class has two class data fields, initialized by the class constructor, and modified by a class destructor, while the `initialization` and `finalization` sections of the unit uses these data fields:

```
class constructor TTestClass.Create;
begin
  StartTime := Now;
end;
```

²⁵ In practical terms, this happens in Delphi 2010 for the new gesturing framework, which is not compiled into the executable if not used.

```

class destructor TTestClass.Destroy;
begin
    EndTime := Now;
end;

initialization
    ShowMessage (TimeToStr (TTestClass.StartTime));

finalization
    ShowMessage (TimeToStr (TTestClass.EndTime));

```

What happens is that the start up sequence works as expected, with the class data already available as you show the information. When closing, instead, the `ShowMessage` call is executed before the value is assigned by the class destructor, which is executed at the very end.

Notice that you can give the class constructor and destructor any name, although `Create` and `Destroy` would be very good defaults. You cannot, however, define multiple class constructors or destructors. If you try, the compiler will issue the following error message:

```

[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class
constructors in class TTestClass: Create and Foo

```

There are a few RTL classes that already take advantage of this new language feature, like the `Exception` class that defines both a class constructor (with the code below) and a class destructor:

```

class constructor Exception.Create;
begin
    InitExceptions;
end;

```

The `InitExceptions` procedure was previously called in the initialization section of the `SysUtils` unit. In general, I think that using class constructors and destructors is better than using unit initialization and termination. In most cases, this is only syntactic sugar, so I won't go back and change existing code. However, if you face the risk of initializing data structures you'll never used (because no class of that type is ever created) moving to class constructors will provide a definitive advantage.

Class Constructors for Generic Classes

A very interesting case arises when you define a class constructor for a generic class. In fact, one such constructor is generated by the compiler and called for each generic class instance, that is, for each actual type defined using the generic template. This is quite interesting, because it would be quite complex to

108 - Chapter 4: More on the Compiler and the RTL

execute initialization code for each actual instance of the generic class you are going to create in your program without class constructor.

As an example, consider a generic class with some class data. You'll get an instance of this class data for each generic class instance. If you need to assign an initial value to this class data, you cannot use the unit initialization code, as in the unit defining the generic class you don't know which actual classes you are going to need.

The following is a bare bones example of a generic class with a class constructor used to initialize the `DataSize` class field, taken from the `GenericClassCtor` example:

```
type
  TGenericWithClassCtor <T> = class
  private
    FData: T;
    procedure SetData(const Value: T);
  public
    class constructor Create;
    property Data: T read FData write SetData;
    class var
      DataSize: Integer;
  end;
```

This is the code of the generic class constructor, which uses an internal string list (see the full source code for implementation details) for keeping track of which class constructors are actually called:

```
class constructor TGenericWithClassCtor<T>. Create;
begin
  DataSize := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

The demo program creates and uses a couple of instances of the generic class, and also declares the data type for a third, which is removed by the linker:

```
var
  genInt: TGenericWithClassCtor <SmallInt>;
  genStr: TGenericWithClassCtor <string>;
type
  TGenDouble = TGenericWithClassCtor <Double>;
```

If you ask the program to show the contents of the `ListSequence` string list, you'll see only the types that have actually been initialized:

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

However, if you create generic instances based on the same data type in different units, the linker might not work as expected and you'll have multiple calls

to the same generic class constructor²⁶ (or, to be more precise, two generic class constructors for the same type). I've added a procedure called `Useless` in the secondary unit of this example that, when uncommented, will highlight the problem, with an initialization sequence like:

```
TGeneri cWi thCl assCtor<System. string>
TGeneri cWi thCl assCtor<System. Smal lInt>
TGeneri cWi thCl assCtor<System. string>
```

Delayed Loading of DLL Functions

In the Windows operating system, there are two ways to invoke an API function of the Windows SDK (or any other DLL): you can let the application loader resolve all references to external functions or you can write specific code that looks for a function and executes it if available. The former code is easier to write, as all you need is the external function declaration, but if the library or even just one of the functions you want to call is not available (a frequent case if your program has to work on multiple versions of the operating system), your program will not be able to start. Dynamic loading allows for more flexibility, but implies loading the library manually, using the `GetProcAddress` API for finding the function you want to call, and invoking it after casting the pointer to the proper type. This kind of code is quite cumbersome, and recent versions of Delphi started adding more and more of it into the VCL, to provide support for Windows Vista (and now Windows 7) features from within applications that still have to work on Windows XP or Windows 2000.

That's why it is good that the Delphi 2010 compiler and linker have added support for a feature now available at the operating system level and already used by some C++ compilers, the delayed loading of functions until the time they are called. The aim of this declaration is not to avoid the implicit loading of the DLL, which takes place anyway, but to allow the delayed binding of that specific function within the DLL.

You basically write the code in a way that's very similar to the classic execution of DLL function, but the function address is resolved the first time the function

26 It is not easy to address a similar problem. To avoid a repeated initialization, you might want to check if the class constructor has already been executed. In general, though, this problem is part of a more comprehensive limitation of generic classes and the linkers inability to optimize them.

is called and not at load time. This means that if the function is not available you get a run-time exception, `EExternal Excepti on`²⁷. However, you can generally verify the current version of the operating system or the version of the specific library you are calling, and decide in advance whether you want to make the call or not.

From the Delphi perspective, the only difference is in the declaration of the external function. Rather than writing (as you can see in the Windows unit):

```
function MessageBox;  
external user32 name 'MessageBoxW';
```

You can now write (again, from an actual example in the Windows unit):

```
function WindowFromPhysical Point;  
external user32  
name 'WindowFromPhysical Point' delayed;
```

At run time, considering that the API has been added to Vista (that is, Windows 6.0) for the first time, you might want to write code like the following taken from the DelayedLoading example:

```
if CheckWin32Version (6, 0) then  
begin  
  hwnd := WindowFromPhysical Point (aPoint);
```

This is nowhere near the amount of code you had to write in previous versions of Delphi to obtain the same behavior. Needless to say that the VCL source code has been significantly updated to use this feature wherever possible, and with the addition of many core API functions that were previously omitted to avoid incompatibilities with older versions of the operating system.

Another relevant observation is that you can use the same mechanism when building your own DLLs and calling them in Delphi, providing a single executable that can bind to multiple versions of the same DLL as long as you use delayed loading for the new functions. Unluckily, the same doesn't apply to packages, but the new Extended RTTI offers enough capabilities for working with packages dynamically that we can also be quite happy about the Delphi 2010 improvements in that area.

²⁷ If you want something more specific and easier to handle at a global level than an exception, you can hook into the error mechanism for the delayed loading call, as explained by Allen Bauer in his blog post: <http://blogs.embarcadero.com/abauer/2009/08/29/38896>

Scoped Enumerators

Although it was introduced in Delphi 2009, scoped enumerations have been hidden enough and I failed to cover them in my “Delphi 2009 Handbook”, so I'm sure it makes sense to cover them here. Traditionally in Delphi the values of an enumeration become global constants you can use freely in your code. This could not be changed, for backward compatibility. The Delphi compiler, however, has a new directive, `$SCOPEDENUMS`, that changes the behavior of enumerations making it compulsory to refer to them with a type prefix²⁸.

Having an *absolute* name to refer to enumerated values removes the risk of a conflict, could let you avoid using the initial prefix of the enumerated values as a way to differentiate with other enumerations, and makes the code more readable, even if much longer to write. Too bad that Code Completion doesn't seem to recognize scoped enumerations, so you actually need to enter the entire type and value manually.

As an example, the `IOUtils` unit (see later in this chapter) defined this type:

```
{ $SCOPEDENUMS ON }
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

This means you cannot refer to the second values as `soAllDirectories`, but you have to refer to it with its complete name:

```
TSearchOption.soAllDirectories
```

This probably sounds quite odd to most Delphi developers, but I guess we'll have to get used to it, whether you like it or not.

The With Statement Now Preserves Read Only Properties

Beside the various extensions covered so far, the compiler in Delphi 2010 has a relevant fix... which can affect existing programs that exploited this unwanted opportunity as if it were a feature. In short, if you have a `with` statement referring to a read-only record property, you cannot modify the record members any more. Again, this is a reasonable fix, but it might affect existing code (and did affect the source of some third party controls). Here is an example, which is the

²⁸ This is exactly how C# invariably works.

ReadOnlyRecord project in the book's source code. Suppose you have a record like the following (the effect is the same for a packed record):

```
type
  TMyPoint = record
    X: Integer;
    Y: Integer;
  end;
```

Suppose you have a class that uses this record as value of a read-only property:

```
type
  TMyFixedRect = class
  private
    fBottomRight: TMyPoint;
    fTopLeft: TMyPoint;
  public
    constructor Create (a, b, c, d: Integer);
    property TopLeft: TMyPoint read fTopLeft;
    property BottomRight: TMyPoint read fBottomRight;
  end;
```

Any code that tries to access the individual element of the read-only record property would not compile in Delphi 2009, nor it does in Delphi 2010:

```
var
  aRect: TMyFixedRect;
begin
  aRect := TMyFixedRect.Create(10, 10, 100, 100);
  aRect.TopLeft.X := 20;
```

However, the following code would have worked in Delphi 2009:

```
with aRect.TopLeft do
  X := 20;
```

In Delphi 2010, this causes a compiler error:

```
[DCC Error] ReadOnlyRecordMainForm.pas(51):
E2064 Left side cannot be assigned to
```

Of course, if you extract the record and modify it, you are modifying a copy, so the original read-only value is not affected, as another code snippet of the demo program demonstrates:

```
aPoint := aRect.TopLeft;
aPoint.X := 20;
ShowMessage (IntToStr (aRect.TopLeft.X));
```

Apparently this change was made because similar code in the context of generic records caused significant problems. While it is a positive change (making the code more robust and clean) it has the drawback of not being backward compatible: Existing Delphi code might fail to compile, although the workaround would probably be reasonably simple.

New Run Time Library Features

Every time the Delphi compiler gets updated, there are extensions of the Run Time Library (RTL) that go along with them. In this release, for example, the System unit defines new types like the `TVisibilityClasses` enumeration and the `TCustomAttribute` class, which complement the Extended RTTI support. There are other trends in the RTL that are worth exploring, beside looking at specific new features and a few brand new units.

RTL Trends

There are a few relevant trends in the Delphi product that show up in the RTL. The first is certainly the **cross-platform trend**. In dozens and dozens of places of the RTL units source code you'll see along with `LINUX` conditional compilation statements (which were never removed from the Kylix²⁹ days) some new `MACOSX` conditional compilation statements. Support for these two platforms is expected in a future version of Delphi. As an example, consider the following definition for the `sLineBreak` global constant:

```
const
  sLineBreak = {$IFDEF LINUX} Ansi Char(#10) {$ENDIF}
               {$IFDEF MSWINDOWS} Ansi String(#13#10) {$ENDIF}
               {$IFDEF MACOSX} Ansi Char(#10) {$ENDIF};
```

Another trend is the focus on **efficient code**. There are countless new inlined functions at the RTL level. Again, as an example, in the `DateUtils` unit (for processing date and time values) 27 existing functions have been marked inline in Delphi 2010.

A third trend is **localization support**: This has been improved by using the operating system `UILocale` property and the complete language-country names (like *fr-FR*) to determine which localized resource DLL to use.

²⁹ Kylix was a version of Delphi for Linux. Not only the compiler could produce Linux executables (limited to the Intel platform), but the entire IDE could run under Linux as host operating system. Despite three versions, Kylix never caught on, also because of some instability of the IDE. That's probably one of the reasons why Embarcadero mentioned its future cross-platform versions of Delphi will use the Windows IDE and let developers cross-compile to other operating systems.

A fourth trend is the slow conversion to code based on **generics** in the RTL. For example, in the System unit there is now the declaration of a generic typed dynamic array, TArray<T>:

```
type
  TArray<T> = array of T;
```

Browsing Existing Units

Beside compiler-related features and trend-setting features, there are many other extensions to the RTL, some of which are worth mentioning:

- The TStringBui lder class has a new Clear method, that removes any data in the current buffer.
- In the Types unit there a new global function called PtInCircle, that clones the existing PtInRect function.
- As already mentioned in Chapter 2, the TThread class has a new static class method to help with naming threads, NameThreadForDebugging.
- Also, the Resume and Suspend methods of the TThread class are now deprecated³⁰. In case you initially create the thread in a suspended state, you should use the new Start method instead of Resume.
 - The TCustomIniFile class has been extended with a new method, ReadSubSections, to improve compatibility with registry access using the class TRegIniFile.
 - The TRegistry class, in turn, has improved error management, with the new properties LastError and LastErrorMsg. Of course, the methods of the class now extensively check for errors, making it much more robust to work with.

There are a couple of other new features that are interesting to notice, as they have quite an ironic spin.

30 The Resume method was deprecated because it was found to be unsafe, leading to memory corruption under certain circumstances. As I learned from Mason Wheeler, *“Basic idea is that Resume un-suspends the thread, and then changes one of the thread's fields, but if you resume and then the thread terminates and frees itself immediately, before the Resume method reaches that point, you're overwriting freed memory which may have been re-allocated already in a high-traffic environment.”*

The first is that the SysUtils unit has a new data type, which can be considered as a *Boolean value with a spin*. Called TUncertainState and used by the Direct2D unit, it is a scoped enumeration³¹ defined as:

```
type
  TUncertainState = (Maybe, Yes, No);
```

The second side note is that Douglas Adams fans should look to the implementation of the GetHashCode_Cl ass function... which uses 42 as return value in given circumstances, rather than causing an exception as it did in the past. I wonder if 42 is just a random non-zero value, or if there is any reason it was picked... other than being the “Answer to the Ultimate Question of Life, the Universe, and Everything”³².

Collections and Containers

Another set of RTL updates relates to collections and containers. The TList class defines a new sorting method, called SortList, which takes an anonymous method as parameter. So in case of a TList storing numbers (as in the btnSortListAnonClick event handler of the RtlLists example, you can write:

```
var
  aList: TList;
begin
  ...
  aList.SortList (
    function (Item1, Item2: Pointer): Integer
    begin
      if Integer(Item1) > Integer (Item2) then
        Result := 1
      else if Integer(Item1) < Integer (Item2) then
        Result := -1
      else
        Result := 0;
    end);
```

The TList class has also three other new methods that let you find the position of an element (or extract it or remove it) starting from the end of the list rather than the beginning. Actually the three methods, listed below, have a TDirection parameter that lets you specify the standard (FromBegin) or reverse (FromEnd) sequence:

31 Scoped enumerations were introduced in first part of this chapter.

32 According to Douglas Adams in “*The Hitchhiker's Guide to the Galaxy*”.

```

function ExtractItem(Item: Pointer;
  Direction: TDirection): Pointer;
function IndexOfItem(Item: Pointer;
  Direction: TDirection): Integer;
function RemoveItem(Item: Pointer;
  Direction: TDirection): Integer;

```

The `RtlLists` project has an example of the usage of `IndexOfItem`, in which the program adds 50 consecutive numbers and then repeats the number one, and you can search for the two occurrences of the value *one* from the beginning or from the end:

```

var
  aList: TList;
  I: Integer;
begin
  ...
  for I := 1 to 50 do
    aList.Add(Pointer(I));
  aList.Add(Pointer(1));
  Log('IndexOf: ' + IntToStr(
    aList.IndexOf(Pointer(1))));
  Log('IndexOfItem (FromEnd): ' + IntToStr(
    aList.IndexOfItem(Pointer(1), FromEnd)));

```

This produces, not surprisingly, the following output:

```

IndexOf: 0
IndexOfItem (FromEnd): 50

```

The same three methods are available also in the inherited `TObjectList`, `TClassList`, and `TComponentList` classes, defined in the `Containers` unit.

These new methods are not part of the generic version of the `TList` class, `TList<T>`, which already had a `LastIndexOf` method since it was introduced in Delphi 2009³³. The generic `TList<T>`, however, has its own four new methods, two for moving items and two for accessing the first and last ones:

```

procedure Exchange(Index1, Index2: Integer);
procedure Move(CurIndex, NewIndex: Integer);
function First: T;
function Last: T;

```

33 I find it quite odd that the method names of these two strictly related classes, `TList` and `TList<T>` are not kept in sync as much as possible. Porting code from `TList` code to the generic version of the container is harder than it should be.

Discovering New Units

In Delphi 2010, the RTL not only has some new capabilities, but also four brand new units. We have already covered one of them, the `Rtti` unit, in Chapter 3. Here come the other three new RTL units (I'll list new Windows API translation headers in a specific section in the next chapter).

The **IOUtils unit** defines a nice set of classes for managing the file system, `TDirectory`, `TPath`, and `TFile`, and will be covered in a following section.

The **TimeSpan unit** defines the `TTimespan` record, similar to the corresponding .NET data structure, and used to express a time difference rather than an absolute time value. The time is stored inside a `TTimespan` in terms of tens of thousandths of a millisecond, that is ten thousands ticks correspond to one millisecond. Compared to a `TDateTime`, which represent the time using the decimal part of a floating point number, a `TTimespan` is way more accurate.

The **Diagnostics unit** defines a handy `TStopWatch` record, which can be used to time an algorithm in a rather precise way, as it uses system ticks and calls the `QueryPerformanceCounter` API to convert ticks to milliseconds in an accurate way. To enable the higher quality measurement, you have to turn on the class property `HighResolution`.

Using the TStopWatch Class

Here is a usage test (from the `StopWatchTest` example):

```
var
  sw: TStopWatch;
begin
  sw := TStopwatch.Create;

  sw.Start;
  // code you want to time
  sw.Stop;

  // read elapsed time
  sw.ElapsedMilliseconds
  sw.ElapsedTicks
```

The first line initializes the frequency counter (depending on the high resolution setting) and zeros the stop watch (as local records are not initialized to zero), the second starts it. As an alternative you can call the combined `StartNew` constructor.

Notice you can Start and Stop the Stop Watch many times in a session: Each time interval will be added. However, you can get the current elapsed time (in ticks, milliseconds, or the new and just introduced `TTimeSpan` structure) also while the Stop Watch is running. With this in mind you can trim the code to:

```
var  
    sw: TStopWatch;  
begin  
    sw := TStopwatch.StartNew;  
    // code you want to time  
    sw.ElapsedTicks
```

The Input/Output Utilities Unit

One of the most interesting additions to the Delphi 2010 Run Time Library, not tied to compiler changes or other new features, is the `IOUtils` unit. This unit has three records mostly defining class methods, which are compatible with the corresponding .NET classes:

- `TDirectory` matches `System.IO.Directory`
- `TPath` matches `System.IO.Path`
- `TFile` matches `System.IO.File`

While it is quite obvious that `TDirectory` is for browsing folder and finding its files and sub-folders, it might not be so clear what is the difference between a `TPath` and `TFile`. The former is used for manipulating file name and directory names, with methods for extracting the drive, file name with no path, extension and the like, but also for manipulating UNC paths. The `TFile` record, instead, lets you check the file time stamps and attributes, but also manipulate a file, writing to it or copying it.

As usual, it can be worth looking at an example. The `IoFilesInFolder` program can extract all of the sub-folders of a given file and it can grab all of the files with a given extension available under that folder.

The program starts by filling an edit box with the Documents folder of the current user. I used the `ShGetFolderPath` API of the `ShlObj` unit (and not that of

the SHFolder unit, available for compatibility with very old versions of Windows³⁴), to get the folder path:

```
procedure TForm1.oFiles.FormCreate(Sender: TObject);
var
  szBufferW: string;
begin
  SetLength (szBufferW, MAX_PATH);
  OleCheck (SHGetFolderPath (Handle,
    CSIDL_MYDOCUMENTS, 0, 0, PChar(szBufferW)));
  edBaseFolder.Text := string (szBufferW);
end;
```

Extracting Subfolders

Of course, you can change that initial folder name to something more appropriate. The program can fill a list box with the list of the folders under that directory, by using the GetDirectories method of TDirectory with the TSearchOption.soAllDirectories³⁵ parameter and enumerating the array of strings that it returns:

```
procedure TForm1.oFiles.btnSubfoldersClick(Sender: TObject);
var
  pathList: TStringDynArray;
  strPath: string;
begin
  if TDirectory.Exists (edBaseFolder.Text) then
    begin
      ListBox1.Items.Clear;
      pathList := TDirectory.GetDirectories(edBaseFolder.Text,
        TSearchOption.soAllDirectories, nil);
      for strPath in pathList do
        ListBox1.Items.Add (strPath);
      end;
    end;
end;
```

Searching Files

A second button of the program lets you get all of the files of those folders, by scanning each directory with a GetFiles call based on a given mask. You can

34 I blogged about the problem of writing this SHGetFolderPath call at http://blog.marcocantu.com/blog/SHGetFolderPath_Default_User.html

35 The IOutils uses scoped enumerators, as covered earlier in this chapter, so you have to prefix the enumerated values with the type name.

120 - Chapter 4: More on the Compiler and the RTL

have more complex filtering by passing an anonymous method of type `TFilterPredicate` to an overloaded version of `GetFiles`.

This example uses the simpler mask-based filtering and populates an internal string list. The elements of this string list are then copied to the user interface after removing the full path, keeping only the file name. As you call the `GetDirectories` method you get only the sub-folders, but not the current one. This is why the program searches in the current folder first and then looks into each sub-folder:

```
procedure TForm1.OFiles.btnPasFilesClick(Sender: TObject);
var
  pathList, filesList: TStringDynArray;
  strPath, strFile: string;
begin
  if TDirectory.Exists (edBaseFolder.Text) then
    begin
      // clean up
      ListBox1.Items.Clear;

      // search in the given folder
      filesList := TDirectory.GetFiles (edBaseFolder.Text, '*.pas');
      for strFile in filesList do
        sFilesList.Add(strFile);

      // search in all subfolders
      pathList := TDirectory.GetDirectories(edBaseFolder.Text,
        TSearchOption.soAllDirectories, nil);
      for strPath in pathList do
        begin
          filesList := TDirectory.GetFiles (strPath, '*.pas');
          for strFile in filesList do
            sFilesList.Add(strFile);
          end;

          // now copy the file names only (no path) to a listbox
          for strFile in sFilesList do
            ListBox1.Items.Add (TPath.GetFileName(strFile));
          end;
        end;
    end;
end;
```

In the final lines, the `GetFileName` function of `TPath` is used to extract the file name from the full path of the file. This is equivalent to using the good old `ExtractFileName` global function. The `TPath` record has a few other interesting methods, which go beyond what was already available in Delphi, including a `GetTempFileName`, a `GetRandomFileName`, a method for merging paths, a few to check if they are valid or contain illegal characters, and much more.

Filtering Sub-folders

There are two problems with the two methods above. First, they have very limited filtering capabilities. When you browse folders for a Delphi file, you'll bump into `__history` folders created for backing up the source code files or sub-folders used by version control systems, which generally start with a period. Also, you can search for one file extension, but not for two at the same time. A second problem is that a search into many sub-folders can be quite slow, so you might want to think of spawning a separate thread to keep the user interface responsive (and provide some clue about the progress).

Using a thread can be a good idea, but both issues can also be fixed using an anonymous method of type `TFilerPredicate`³⁶. This version of the `btnSubFoldersClick` method solves the first problem:

```
procedure TForm1.Ofiles.btnFilterFoldersClick(Sender: TObject);
var
  pathList: TStringDynArray;
  strPath: string;
begin
  pathList := TDirectory.GetDirectories(edBaseFolder.Text,
    TSearchOption.soAllDirectories,
    function (const Path: string;
      const SearchRec: TSearchRec): Boolean
    begin
      Result := not (SearchRec.Name = '__history') and
        not (SearchRec.Name[1] = '.');
    end);
  for strPath in pathList do
    ListBox1.Items.Add (strPath);
  end;
```

The final version differs only in the anonymous method and addresses both issues:

```
pathList := TDirectory.GetDirectories(edBaseFolder.Text,
  TSearchOption.soAllDirectories,
  function (const Path: string;
    const SearchRec: TSearchRec): Boolean
  begin
    Result := not (SearchRec.Name = '__history') and
      not (SearchRec.Name[1] = '.');
    Inc (nTotal);
    if Result then
      Inc (nFound);
```

36 If you don't like the embedded anonymous methods definition syntax, you can create a separate function and pass it as a parameter, although this partially defeats the notion of using anonymous methods (that is, their ability to capture the execution context).

```

        StatusBar1.SimpleText := Format (
            'Folders %d/%d', [nFound, nTotal]);
    Application.ProcessMessages;
end);

```

Filtering Files

In a very similar fashion, we can write a filter looking for both Pascal source code files (. pas) and Delphi project files (. dpr). At the core of the new method, which again parses folders removing __history and folders starting with a period, there is the following GetFiles call with a filter predicate:

```

filesList := TDirectory.GetFiles (strPath, '*.*',
    function (const Path: string;
        const SearchRec: TSearchRec): Boolean
    var
        strExt: string;
    begin
        strExt := TPath.GetExtension(SearchRec.Name);
        Result := (strExt = '.pas') or (strExt = '.dpr');
        Inc (nTotal);
        if Result then
            Inc (nFound);
        StatusBar1.SimpleText := Format (
            'Files %d/%d', nFound, nTotal);
        Application.ProcessMessages;
    end);

```

The three record structures in the IOUtils unit have many more features than I've covered and are certainly worth a second look. File operations in Delphi used to be low level, unless you used a third-party library. In Delphi 2009, the development team fixed text file manipulation with the new TStreamReader and TStreamWriter classes, now in Delphi 2010 there is also a higher level approach for manipulating folders, paths, file names and properties.

What's Next

Over the last two chapters I focused on new features of the compiler and the RTL, the core foundations of all Delphi applications. Most programs written with Delphi, however, also have a user interface built using the VCL and interact with the operating system. In the next three chapters I'll focus on new VCL features and on the support for the latest version of the OS, Windows 7.

Chapter 5: The VCL And Windows 7

If Delphi's Run Time Library lets you perform some basic operations, it is the Visual Component Library (VCL) that provides the core interaction with the Windows operating system and lets you create the user interface of your programs. In Delphi 2010 the VCL was updated to support the latest version of the operating system, Windows 7.

In this chapter I'll explore new features of Windows 7, the available support in the VCL, and other new VCL extensions. I'll also show you how to program against Windows 7 in areas for which the VCL has no explicit support.

Tech Overview of Windows 7

A lot can be said about Windows 7 and I certainly have no room for a full evaluation of this new operating system. My aim in this short section is only to give you some technical information about changes compared to Windows Vista.

For of all, notice that the version of the operating system is not 7 (as you might expect) but 6.1. This can be easily verified by using the Delphi global variables `Win32MajorVersion` and `Win32MinorVersion`.

If you need to check if the version of Windows your program is running onto is at least Windows Vista or Windows 7, you can use the `CheckWin32Version` function as follows³⁷:

```
if CheckWin32Version(6) then
  Log ('Running at least on Vista');
if CheckWin32Version(6, 1) then
  Log ('Running on 7');
```

Beside many improvements, in terms of a reduced memory footprint and a more functional user interface, Windows 7 has a significant number of new API³⁸ functions. It is relevant to notice that these new APIs come in two forms: C-language functions and COM interfaces. That is, the Windows API is being extended in a very traditional way, that doesn't require programming to be with .NET. On the contrary, Delphi offers complete support for all of the new features of the operating system.

To a very large extent, Windows 7 is very similar to Windows Vista, having the same driver model, the same User Account Control and Windows Resource Protection mechanism, the same desktop windows management, the same Glass theme activated by default, and much more. Still, there are areas in which there are distinct differences, from the behavior of Taskbar buttons to the introduction of Libraries, from an extended version of DirectX to new graphical

37 This code is part of the `GetOSVersion` example, updated from the original version of the demo in my *Delphi 2007 Handbook*.

38 The Windows API, or Application Programming Interface, is the collection of all of the functions of the operating system that an application can call. The Windows API is huge and offers countless operations, built around the three libraries that still constitute the core of the operating system (from the Windows 1.0 days), Kernel, User and GDI. The Windows API is fully documented in the SDK help that ships with Delphi and is also on the MSDN web site, <http://msdn.microsoft.com/en-us/library/default.aspx>.

processing components, from gestures and touch support to an inertia processor, from a sensors platform to federated search support.

While some of these new features are well integrated into the VCL in Delphi 2010, others require using the API directly, something that should not cause any great problem to Delphi developers. I'll delve into some of these new features of the operating system throughout this chapter and the next.

It might be worth starting by providing a summary of the support already available for Vista (and hence also for Windows 7, for those migrating to it directly from Windows XP) in the last two versions of Delphi. Details of these features were covered in my two most recent Delphi Handbooks.

Delphi Support for Windows Vista

Starting with Delphi 2007, the Delphi R&D team has been putting considerable effort into supporting new versions of the operating system in the VCL. This is a change from the past, as in the name of compatibility with older versions of the operating system, Delphi 2006 still didn't provide explicit support for many features of Windows XP like new control styles.

Needless to say that in some cases taking advantage of new features of Vista might mean that your application will not run as expected on Windows XP or Windows 2000³⁹. Here is a very condensed summary of recent Delphi features specifically focused on supporting Windows Vista:

- The `DefaultFont` property of the `Application` object has been added to simplify the support for the new system font, Segoe UI, in Vista applications. This font is used by default by all forms of your application (if their `ParentFont` property is set to `True`). There is also a similar `MessageFont` property in the `Screen` global object, used for the font of some of the message dialogs.
- In Windows Vista traditional Delphi applications had problems displaying the preview of minimized application in the Windows Flip (or Taskbar previews), Windows Flip 3D, and the Alt-Tab window. To overcome this issue, the `Application` object has a new `MainFontOnTaskbar` property that enables the display of the main form, rather than the application hidden

³⁹ Support for Windows 95 and 98 at target platforms for Delphi applications was dropped in Delphi 2009, with the advent of Unicode. Platforms with very limited Unicode support couldn't be supported any more.

window, in the Taskbar. As I'll discuss in the next section, this is no longer needed in Windows 7.

- Forms have a new `GlassFrame` property that lets you easily extend the glass frame of windows into their client area. Many components have been extended to support being painted over the glass surface, although you might have to enable their `DoubleBuffered` property to get correct output.
- Vista Task dialogs are supported by Delphi in two different ways. There is a new specific `TaskDialog` component, with extended features, and a simple `TaskMessageDlg` function that gets automatically called in place of `MessageDlg` when you set the global `UseLatestCommonDialogs` to `True`.
- The new Vista Open and Save dialog boxes (implemented by the `IFileOpenDialog` and `IFileSaveDialog` interfaces) are directly mapped by the new `FileOpenDialog` and `FileSaveDialog` components, but also the standard `OpenDialog` and `SaveDialog` component uses the new style when the global `UseLatestCommonDialogs` is set.
- Themes can now be enabled through a project options setting, which also activates them at design time. Themes are more relevant than in the past and many new user interface features in Vista (and Windows 7) are available only to themed applications.
- The new `TreeView` style with small triangles replacing the plus and minus symbols is enabled by default in the VCL.
- Button controls have been enhanced with support for the command link and split button styles.
- Label controls support the glowing style when painted on a glass surface.
- Several styles already available since Windows XP but not part of the VCL for a long time have recently been implemented. These new VCL features include extensions to edit boxes (text hints, numbers filtering, and alignments), grouping support for the `ListView` control, and `RichEdit` version 2 support.
- Progress bars have been extended to support the newest options, including status information (affecting the progress bar color), smooth movements, and smooth reverse movements. Some of these features were already in Windows XP, while others are specific to Vista.
- `ShellResources` is a new Delphi component that helps you overcome the lack of the standard system animations in Vista and beyond. If you are using an `Animate` common control with the standard animations that were available in older versions of Windows, your program won't work any more as those animations are no longer part of the operating system. As an easy fix, add

this component anywhere in your program (or simply include its unit) to compile resources corresponding to the animations that used to be in the operating system into your own application.

As you can see from this list, which is far from complete and detailed, recent versions of Delphi provide a significant amount of support for new features of the recent versions of the Windows operating system. This has been a significant change as in the past backward compatibility with Windows 9x has limited the support for similar extensions.

Delphi 2010 pushes this support one step forward, with new specific components, new API declarations (also thanks to delayed loading⁴⁰), and the Object Pascal version of many new COM interfaces. Before delving into this topic, however, let me focus on changes between Vista and Windows 7 that affect some of the features discussed earlier and covered in my previous books.

Notable Differences Between Vista and Windows 7

Having covered in summary new features of recent version of Delphi that let you better support Windows Vista and Windows 7, I have to underline the fact that Windows 7 adds a few extra features that make existing / old Delphi applications more compatible with the new operating system. There are a couple of these differences worth mentioning.

The first relates with the preview of the application main form available in Windows Flip (the task bar preview), Windows Flip 3D, or even the plain list of windows you obtain using Alt+Tab keys. In Vista, a traditional Delphi application would have been represented by its icon when it was minimized and displayed in any of these views⁴¹.

The fix came to the VCL in Delphi 2007 with the `MainFormOnTaskbar` property of the `TApplication` class. In Windows 7, however, a traditional Delphi application would show properly in the various previews, even when minimized. This basically means that the `MainFormOnTaskbar` property becomes

40 Delayed loading of DLL functions was covered in Chapter 4.

41 This feature of Vista and the problems it poses, along with possible solutions is discussed in details in my Delphi 2007 Handbook. The same is true for the Windows Resource Protection problems covered next.

much less relevant, although it will still affect the title displayed for the application in the taskbar. With older versions of Delphi, or in case the property is set to `False`, the title will match the `Title` property of the `Application` global object; on the other hand, if `MainForm.OnTaskbar` is set to `True`, the title is the `Caption` of the main form.

Another relevant change relates to the behavior of Windows Resource Protection and the Virtual Storage (the area created for each user to host their document and configuration files that old “non-themed” applications save in Program Files sub-folders or in the Windows folder). Windows 7 expands the virtual storage area to include the root of the C: drive⁴².

As an example, the `FileAccess` program discussed in my “Delphi 2007 Handbook” tried to save a file to the root of the C: drive with the code:

```
procedure TFormFile.btnSaveRootClick(Sender: TObject);
begin
    Memo1.Lines.SaveToFile('C:\SomeText.txt');
end;
```

In Vista this code used to fail with an error both for a themed and a non-themed application, in Windows 7 the themed application succeeds and saves the file to the basic folder of the virtual store area, which on my computer (for my account) is:

```
C:\Users\Marco\AppData\Local\Virtual Store
```

The full source code of the program is available in the `FileAccess` folder. Notice there are two similar applications, compiled with and without a manifest (that is, themed or non-themed).

By the way, compared to the version written for Delphi 2007, I had to update the `RunAsAdmin` procedure I originally borrowed from Fredrik Haglund, to support wide strings and the call of the 'wide' version of `ShellExecuteEx`.

This is the code used to ask for elevation of the child process:

```
procedure RunAsAdmin(hWnd: HWND; aFile: string;
    aParameters: string);
var
    sei: TShellExecuteInfo;
begin
    FillChar(sei, SizeOf(sei), 0);
    sei.cbSize := sizeof(sei);
```

⁴² I've noticed that in some circumstances this helps BDE applications to behave better on Windows 7 than on Windows Vista, but you still have to redefine the `NET` dir and... get rid of the BDE as soon as you can! Just a personal opinion, of course.

```

sei.Wnd := hWnd;
sei.fMask := SEE_MASK_FLAG_DDEWAIT or
SEE_MASK_FLAG_NO_UI;
sei.lpVerb := 'runas';
sei.lpFile := PChar(aFile);
sei.lpParameters := PChar(aParameters);
sei.nShow := SW_SHOWNORMAL;

if not ShellExecuteEx(@sei) then
  RaiseLastOSError;
end;

```

Delphi 2010 Windows API Units

As I mentioned earlier, Delphi 2010 piggy backs on the support available in Delphi 2009 for Windows Vista, completing it with specific extra features. Before getting to new VCL components supporting Windows 7, it is worth looking at the first level of the Windows SDK support, that is Windows API import units and COM interface declarations.

Beside brand new units, there are also quite a few that have been extended in significant ways, with the inclusion of many new APIs or COM interfaces.

New API Header Units

Several new RTL units have been added to Delphi 2010 to provide the Pascal language translation of the header files of new libraries of the Windows SDK. These new libraries include DirectX support and several others.

Although I don't have room to delve into them in detail, here is a partial list of the new API header units, partially coming from third parties (and available with the MPL open-source license) and partially introduced by the Delphi R&D team for the new version of the product:

- There are new headers for **interfacing .NET** from unmanaged code, specifically hooking into `mscorlib.dll`. These are provided in the `Cor` unit, along with two support units, `CorError` and `CorHdr`.

- **DirectX support headers developed by Project JEDI⁴³** are now included in the core Windows units, comprising:

Direct3D	Direct3D8	Direct3D9
DirectDraw	DirectInput	DirectMusic
DirectPlay8	DirectSetup	DirectShow9
DirectSound	DxDiag	DXFile
DXTypes	DX7toDX8	D3DX8
D3DX9		

- **Other DirectX header types**, related with technologies introduced in Windows Vista and Windows 7 and partially wrapped by the VCL are available the following new units:

D2D1, core elements for Direct2D support, some of which are covered later in the section “Direct2D”.

DxgiFormat, headers for the core DirectX Graphics Infrastructure⁴⁴.

WMF9, Windows Media Format 9 API (conversion provided by Henri Gourvest).

Manipulations, hosts the interface for the inertia manipulation engine of Windows 7, used for moving things around your screen in a more realistic way (there is an example of the use of the manipulation and the inertia processors in the next chapter).

Wincodect provides headers for interfacing windowscodecs.dll and is used to enable support for TWICImage, covered later in this chapter in the section “Using Windows Imaging Component”.

- The definition of the **IObjectArray** and **IObjectCollection** interfaces (used among others by some of the Shell APIs) is now available in the new **ObjectArray** unit.

43 The Joint Endeavour of Delphi Innovators (Project JEDI, <http://www.delphi-jedi.org>) is one of the most well known and active Delphi open source initiatives, which started off by translating API headers not found in the product. Offering access to the DirectX API was part of this task. Most of the work on the DirectX headers has been done by Alexey Barkovoy, but other contributors are listed in the units themselves.

44 See among other sources http://en.wikipedia.org/wiki/DirectX_Graphics_Infrastructure and [http://msdn.microsoft.com/en-us/library/bb205075\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb205075(VS.85).aspx)

- **Windows Search**⁴⁵ support is provided by the interfaces to the structured query interfaces located in the units `StructuredQuery` and `StructuredQuery-Condition`.
- There are other new headers for **tablet support**, including `TpcShrd`, `RtsCom`, and `MsInkAut`.

Extended Windows API Headers

Some of the existing library files, starting with the base Windows unit, have also been augmented with new functions. Some of these API functions were simply missing, while others were operating-system version dependent, and have now been implemented using the new `delayed` directive⁴⁶.

Some noteworthy additions to **Windows unit** include APIs that have been around for some time, but were not previously available in the core Delphi interface unit for the Windows API:

```
function GetLongPathName(lpShortPath: PWideChar;
  lpLongPath: PWideChar; cchBuffer: DWORD): DWORD; stdcall;
function GetProcessHandleCount(hProcess: THandle;
  var pdwHandleCount: DWORD): BOOL; stdcall;
function GetProcessId(Process: THandle): DWORD; stdcall;
function GetComputerNameEx(NameType: TComputerNameFormat;
  lpBuffer: PWideChar; var nSize: DWORD): BOOL; stdcall;
```

Two interesting new functions (part of the SDK since a service pack of Windows XP) are `SetDllDirectory` and `GetDllDirectory`, which you can use to determine and change the folder from which an application will load its dynamic libraries and Delphi run-time packages.

There are about new 10 “timer queue” support functions, starting with:

```
function CreateTimerQueue: THandle; stdcall;
```

There are also various touch input support functions and data structures, gesture support, devices integration, physical points support and the like.

45 For a reference to the Windows Search SDK you can refer to [http://msdn.microsoft.com/en-us/library/aa965362\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa965362(VS.85).aspx)

46 Delayed functions are used to avoid the standard DLL load time binding, which might prevent an application to run on an older version of the operating system even for a single extra API call. This was described in the section “Delayed Loading of DLL Functions” of Chapter 4.

Most of the API functions that are specific to Windows Vista or Windows 7 are declared as *delayed*, including:

```
// Vista desktop management
CreateDesktopEx
GetIconInfoEx
SetProcessDPIAware

// Vista logical cursors
GetPhysicalCursorPos and SetPhysicalCursorPos
LogicalToPhysicalPoint and PhysicalToLogicalPoint
WindowFromPhysicalPoint

// Vista power management
RegisterPowerSettingNotification
UnregisterPowerSettingNotification

// Vista clipboard notifications
AddClipboardFormatListener
RemoveClipboardFormatListener
GetUpdatedClipboardFormats

// Windows 7 display management
CalculatePopupWindowPosition
GetWindowDisplayAffinity and SetWindowDisplayAffinity

// Windows 7 touch support
GetTouchInputInfo
IsTouchWindow
CloseTouchInputHandle
RegisterTouchWindow and UnregisterTouchWindow

// Windows 7 gestures support
GetGestureInfo and CloseGestureInfoHandle
GetGestureExtraArgs
SetGestureConfig and GetGestureConfig
```

The companion **Messages unit** has the definition for new messages (including `wm_Gesture`, `wm_Touch`, `wm_ClipboardUpdate`, and a few `wm_XButton` ones) and their related data structures.

The **DwmApi unit** has been largely extended and rewritten using delayed loading rather than custom dynamic loading. There are also some new functions added for Windows 7. This unit interfaces the Desktop Windows Manager⁴⁷ library first introduced in Vista.

The **ShlObj unit** has been largely extended, providing the definition of dozens of new interfaces, some of which were already part of previous versions of Win-

⁴⁷ See <http://msdn.microsoft.com/en-us/library/aa969540.aspx> for a detailed introduction to the Desktop Windows Manager API.

dows, while others have been added for Windows Vista and Windows 7. The latter include the `I Taskbar` and the `I Shell Library` interfaces I'll demonstrate later in this chapter in the section "Working with Taskbar Buttons in Windows 7". To have an idea of how many more interfaces are covered, consider this unit has grown from 4,000 lines to over 14,000 lines (or from 185 Kb of source code to over 667 Kb). Also the `ShellAPI` unit sees a significant revision, incorporating new and missing API functions and interfaces.

The **WinSpool** unit (hosting the Win32 printer API Interface) is now almost twice as large and supports many newer capabilities of the operating system.

Finally, the **mxsml** unit has extended support for version 6 of the Microsoft XML DOM engine. I'll cover the changes in XML support in Delphi 2010 in Chapter 8.

Windows 7 Support

Having looked at a rather long list of changes in terms of low-level API support, it is worth considering which new features of Windows 7 we can program against using some of these new APIs. Later I'll move to features specifically supported by the VCL, with ready-to-use components, including the `Direct2D` support and more.

Two of the new features of Windows 7 that are most visible to users are the changes to taskbar buttons (which can provide status information about the running application and direct commands to interact with it) and the introduction of libraries (a new way to collect and manage files and folders).

Working with Taskbar Buttons in Windows 7

One of the most noticeable new features of the Windows 7 user interface is the new role of taskbar buttons, the graphic generally positioned at the bottom of the screen and showing the various applications that are currently running. In Windows 7 you can mix running applications with regularly used ones: You *pin* a program to the taskbar and its icon will remain there even when the application is not running.

These new taskbar buttons provide several ways to interact with an application. When the program is closed, you can see recent documents opened with it and see a menu with custom operations. When the program is running you can see a preview of its main windows (fully customizable), see status information, add extra buttons and commands, see the progress of a slow operation, and much more. I will not explore all of the features of taskbar buttons in Windows 7 here, while building an example called Win7Taskbar. I'll cover only the most common taskbar features⁴⁸, while showing you how to use the related COM interfaces, which are part of the Windows Shell API.

The TaskList Interfaces

The ShlObj unit in Delphi 2010 defines the interfaces you can use to interact with the taskbar elements. There are actually four of these interfaces, each extending the previous one:

```
I TaskbarLi st = i nterface(I Unknown)
I TaskbarLi st2 = i nterface(I TaskbarLi st)
I TaskbarLi st3 = i nterface(I TaskbarLi st2)
I TaskbarLi st4 = i nterface(I TaskbarLi st3)
```

What you can do is to ask the system for an object implementing taskbar support and extract the various interfaces from it. As a helper you can use across projects, I've defined the following data structure:

```
type
  TTaskBarSupport = class
  public
    TaskbarLi st: I TaskbarLi st;
    TaskbarLi st2: I TaskbarLi st2;
    TaskbarLi st3: I TaskbarLi st3;
  public
    constructor Create;
    procedure Ini tTaskbarSupport;
  end;
```

The core of this class is in the method that creates the first COM objects and extracts its various interfaces⁴⁹:

```
procedure TTaskBarSupport. Ini tTaskbarSupport;
begin
  TaskbarLi st := CreateComObj ect(CLSI D_TaskbarLi st)
  as I TaskbarLi st;
```

48 For more examples and Delphi components you can use to simplify your interaction with the Windows 7 taskbar (also using older versions of Delphi) see Daniel Wischniewski blog on www.gumpi.com and (in particular) the post: <http://www.gumpi.com/Blog/2009/09/27/DelphiControlsForWindows7StateUpdate.aspx>

```

TaskbarLi st.HrI ni t;
Supports(TaskbarLi st, IID_I TaskbarLi st2, TaskbarLi st2);
Supports(TaskbarLi st, IID_I TaskbarLi st3, TaskbarLi st3);
end;

```

The TaskbarSupportUnit unit that defines this data structure creates a global instance of the class (called TaskBarSupport) in its initialization section, so it can be used directly within any program that includes the unit.

A Progress in the Taskbar

Now that we have a way to access to the taskbar interfaces, we can start implementing one of the new Windows 7 taskbar features, namely the display of progress bar information within a taskbar button, as you can see here on the right. This display is enabled by calling the ITaskbarLi st3.SetProgressState method, while the actual progress is set by calling ITaskbarLi st3.SetProgressVal ue. At the end you have to restore the progress state of the taskbar button to normal.



To make the user interface more intuitive, I've also added a ProgressBar component to the main form, moving it in sync with the taskbar button:

```

procedure TWi n7TaskForm. btnProgressCl ick(Sender: TObj ect);
var
    I: Integer;
    FormHandl e: THandl e;
begin
    FormHandl e := GetTaskBarEntryHandl e;
    TaskbarSupport. TaskbarLi st3. SetProgressState(
        FormHandl e, TBPF_NORMAL);
    for I := 1 to 100 do
        begin
            ProgressBar1. Posi ti on := I;
            TaskbarSupport. TaskbarLi st3. SetProgressVal ue(
                FormHandl e, I, 100);
            Appl i cati on. ProcessMessages;
            Sl eep (100);
        end;
    TaskbarSupport. TaskbarLi st3. SetProgressState(
        FormHandl e, TBPF_NOPROGRESS);
end;

```

-
- 49 Notice that the class ID of the COM object you pass to the CreateComObj ect function (in this case CLSI D_TaskbarLi st) matches the interface ID of the interface the object implements (in this case IID_I TaskbarLi st). I find this approach confusing, as it doesn't follow standard COM development guidelines, but you can easily get used to it.

Notice that each of the calls requires the handle of the form visible in the taskbar. This can be either the main form or the application hidden form, depending on the VCL configuration:

```
function GetTaskBarEntryHandle: THandle;
begin
  if not Application.MainFormOnTaskBar then
    Result := Application.Handle
  else
    Result := Application.MainForm.Handle;
end;
```

When trying this feature notice that if the application is not active, the progress bar in the taskbar icon will have a more contrasted color... and be more visible.

Overlay Icons

A second taskbar button feature you can work with is overlay icons. These are small icons you can add on top of the icon in the taskbar button



(that is the form icon, by default). Here on the right you can see the original icon and a couple of overlay elements. These elements, in my demo program, are extracted from an ImageList using the following code:

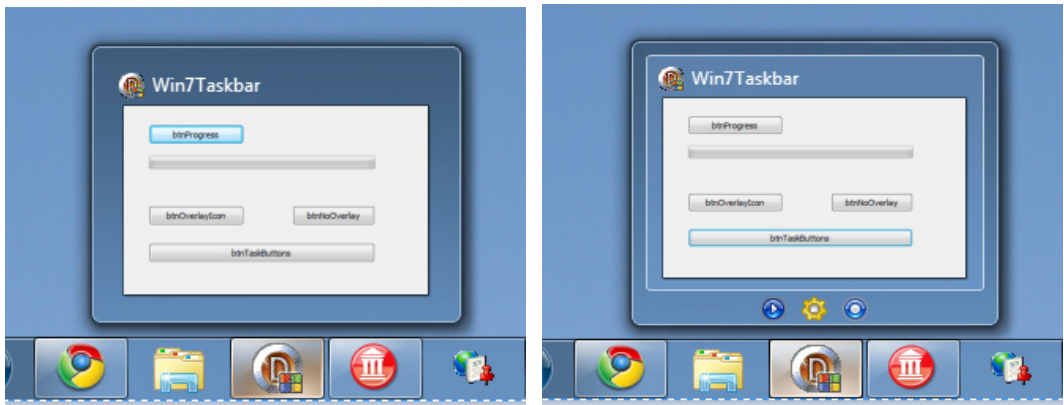
```
procedure TWi n7TaskForm. btnOverl ayIconCl ick(
  Sender: TObje ct);
var
  anIcon: TIcon;
begin
  anIcon := TIcon. Create;
  try
    ImageLi st1. GetIcon(Random(3), anIcon);
    TaskbarSupport. TaskbarLi st3. SetOverl ayIcon(
      GetTaskBarEntryHandle, anIcon. Handle,
      PChar('MyIcon'));
  finally
    anIcon. Free;
  end;
end;
```

The `TaskbarLi st3. SetOverl ayIcon` method requires the form handle, the handle of the icon, and a description for accessibility purposes as parameters. To remove the current overlay icon, pass 0 for the icon handle, as the program does in the `btnNoOverl ayCl ick` method.

Task Buttons

The last feature I want to demonstrate in the Win7Taskbar example is the support for task buttons. These are small buttons that get displayed along with the miniature of the application's main form as you move the mouse over the taskbar entry. You can associate code with the click event on those buttons, by handling the `wm_SysCommand` message. Before I get to the code of the program, though, it is worth showing you a picture of task buttons.

Below you can see a regular version of the miniature view of the demo program compared to a version with three task buttons activated:



Now that you have seen what the user interface for taskbar buttons looks like, we can see how to activate it in your Delphi code. What you have to do is create an array of `TThumbButton` elements, give each of them an id, the number of an element from an `ImageList`, set a few masks and flags, and provide a hint. At the end you'll pass this array (technically the address of its first element) as parameter to the `ITaskbarList3.ThumbBarAddButtons` method after assigning the `ImageList` to the taskbar button:

```
procedure TWin7TaskForm.btnTaskButtonsClick(Sender: TObject);
var
  Buttons: array of TThumbButton;
  I: Integer;
begin
  SetLength(Buttons, 3);
  for I := 0 to 2 do
  begin
    Buttons[I].iId := I;
    Buttons[I].iBitmap := I;
    Buttons[I].dwMask := THB_FLAGS or THB_BITMAP or THB_TOOLTIP;
    Buttons[I].dwFlags := THBF_ENABLED or
```

```

    THBF_NOBACKGROUND or THBF_DISSIMILARCLICK;
    StrCopy (Buttons[1].szTip,
    PChar('button ' + IntToStr (1)));
end;
TaskbarSupport.TaskbarList3.ThumbBarSetImageList(
    GetTaskBarEntryHandle, ImageList1.Handle);
TaskbarSupport.TaskbarList3.ThumbBarAddButtons(
    GetTaskBarEntryHandle, Length(Buttons), @Buttons[0]);
end;

```

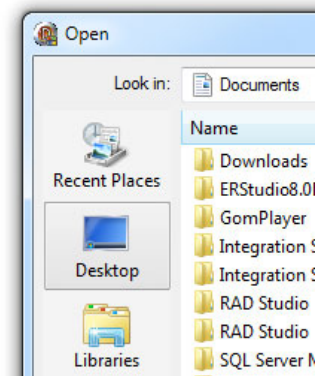
Working with Libraries

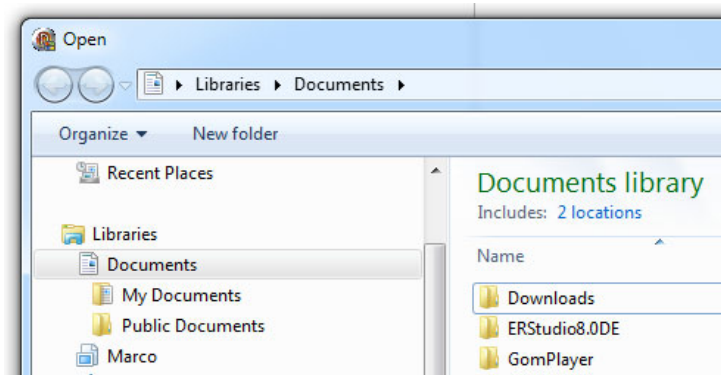
Another specific new feature of Windows 7 is the definition of libraries. These are collections of files and folders groups under a custom name and easier to reach even if they reside in different areas of your hard drive, external drive, network drive, and so on.

I don't want to delve into the behavior of libraries in Windows 7 here or the reason for using them (although I have to say that I like the idea a lot), but focus on how you can interact with the libraries from a Delphi application. The two most relevant interfaces to use, both declared in the `ShlObj` unit, are `IShellLibrary` and `IShellItem`. You might also want to use some of the default folder constants declared in the `KnownFolders` unit.

Before looking at these interfaces, though, let's look at how libraries appear in a classic `OpenDialog` and a new `FileOpenDialog`. To demonstrate this the `Win7Libraries` example has both components and two buttons to active them. In case of the traditional dialog box, the program disables the `UseLatestCommonDialog` global variable to disable the automatic redirection of the old common dialog box to the newer version.

In the classic `OpenDialog`, libraries are just a side icon, as you can see here on the right. In the newer `FileOpenDialog`, instead, the navigational tree includes libraries and the ability to expand their details, as you can see in the next image.





There are several ways to interact with libraries and their information. First of all, you can query the system for the file defining the library:

```
var
  pch: PChar;
begin
  OleCheck(SHGetKnownFolderPath (
    FOLDERID_DocumentsLibrary, 0, 0, pch));
  Memo1.Lines.Add('SHGetKnownFolderPath: ' + pch);
```

In this case the name of the file physically hosting the library information, returned by the code snippet above, would be:

```
SHGetKnownFolderPath: C:\Users\Marco\AppData\Roaming\Microsoft\
  Windows\Libraries\Documents.Library-ms
```

If you want to access to more details of the given library, you can use the `SHGetKnownFolderPathItem` function, which fills a pointer with an `IShellItem` interface:

```
var
  item: IShellItem;
  pch: PChar;
  pnt: Pointer;
begin
  OleCheck(SHGetKnownFolderPathItem(
    FOLDERID_DocumentsLibrary, 0, 0, IShellItem, pnt));
  item := IShellItem(pnt);
  item.GetDisplayName(SIGDN_NORMALDISPLAY, pch);
  Memo1.Lines.Add('SHGetKnownFolderPathItem.GetDisplayName: ' + pch);
```

The output in this case would be:

```
SHGetKnownFolderPathItem.GetDisplayName: Documents
```

To access the actual library and manipulate it, we need to perform a further step and ask the system for a COM object implementing the `IShellLibrary` interface that we'll later initialize with the proper `IShellItem` element:

```

var
  aLibrary: IShellLibrary;
  Obj: IInterface;
  item: IShellItem;
  pch: PChar;
  pnt: Pointer;
begin
  OleCheck(SHGetKnownFolderItem(
    FOLDERID_DocumentsLibrary, 0, 0, IShellItem, pnt));
  item := IShellItem(pnt);
  aLibrary := CreateComObject(CLSID_ShellLibrary) as IShellLibrary;
  aLibrary.LoadLibraryFromItem(item, PropSys.GPS_READWRITE);

```

Now that we have the library interface tied to the `IShellItem` for the given folder, we can use it, for example to extract the list of elements it contains:

```

var
  ...
  anArray: IObjectArray;
  nItems: Cardinal;
  I: Integer;
begin
  ...
  aLibrary.GetFolders(LFF_FORCEFILESYSTEM, IObjectArray, anArray);
  anArray.GetCount(nItems);
  for I := 0 to nItems - 1 do
    begin
      anArray.GetAt(I, IShellItem, item);
      item.GetDisplayName(SIGDN_NORMALDISPLAY, pch);
      Memo1.Lines.Add('Library item: ' + pch);
    end;

```

In the specific case this produces the list of the folders in library:

```

Library item: My Documents
Library item: Public Documents

```

Other operations you can accomplish on a library include adding and removing folders, getting and changing the default save folder for the library, reading and changing the folder type (music, documents, and the like), and extracting or changing the library icon.

The demo program has a further feature it might be worth exploring. It let's you open and display the XML file that defines the library, using this code:

```

procedure TForm1.btnLibraryXmlClick(Sender: TObject);
var
  pch: PChar;
begin
  OleCheck(SHGetKnownFolderPath (
    FOLDERID_DocumentsLibrary, 0, 0, pch));
  Memo1.Lines.LoadFromFile(pch);
end;

```

The result is something like the following, not exactly a readable description even if it uses the XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<libraryDescription
  xmlns="schemas.microsoft.com/windows/2009/library">
  <isLibraryPinned>true</isLibraryPinned>
  <iconReference>imagres.dll,-1002</iconReference>
  <searchConnectorDescriptionList>
    <searchConnectorDescription>
      <isDefaultSaveLocation>true</isDefaultSaveLocation>
      <simplerLocation>
        <url>knownfolder: {FDD39AD0-...}</url>
        <serialized>...binary data...</serialized>
      </simplerLocation>
    </searchConnectorDescription>
    <searchConnectorDescription>
      <isDefaultNonOwnerSaveLocation>true
      </isDefaultNonOwnerSaveLocation>
      <simplerLocation>
        <url>knownfolder: {ED4824AF-...}</url>
        <serialized>...binary data...</serialized>
      </simplerLocation>
    </searchConnectorDescription>
  </searchConnectorDescriptionList>
</libraryDescription>
```

The folders of this library are all standard ones, so they are listed using known folders GUIDs (the two elements in bold). I doubt you want to process such a file manually, so using the API discussed earlier seems the best approach.

DirectX for Forms

The DirectX libraries have been part of the Windows API for a long time, but have always been regarded as a separate user interface model, mostly good for games or high-performance 3D graphics. In Windows 7 for the first time, some of the features of DirectX can be used by standard applications, as happens in the .NET Framework when using the Windows Presentation Foundation, WPF.

Windows Vista introduced a new infrastructure for DirectX called Windows Display Driver Model (WDDM), which lets multiple applications share the services of the Graphical Processing Unit (so that DirectX can also be used by non full-screen applications). This feature was actually used in Vista by the Desktop Windows Manager to display Flip3D and the Aero Glass effect. A further big

change comes with Windows 7, which makes available a new set of DirectX APIs that you can use to let standard applications display 2D and 3D graphic, text, and images in a much faster way. Microsoft has promised to make Direct2D available also as a Vista update⁵⁰.

One of the changes, for example, is the ability to use one of the Direct2D libraries for painting a standard form or visual component. You can even mix standard GDI output (based on the `TCanvas` class in the VCL) and the new Direct2D output (for which you can use the new `TDirect2DCanvas` class), as we'll see in practice in the next section.

Direct2D

The native VCL support for Direct2D is based on an abstraction of the existing `TCanvas` class. The Graphics unit defines a new base class, called `TCustomCanvas`, from which `TCanvas` now inherits along with the new `TDirect2DCanvas` class (which in turn is defined in the new Direct2D unit). For compatibility purposes, most of the existing methods of `TCanvas` also still work with the new Direct2D class.

There are a few extra methods, but you access most of the specific Direct2D features by using the `RenderTarget` interface (of type `ID2D1RenderTarget`). This acts like a handle for the `TCanvas`, but it is a COM interface providing ready to use methods rather than being a handle you can pass to specific API functions. Other low-level features are accessible using the `D2DFactory` and `WriteFactory` methods of the `TDirect2DCanvas` class.

This is why if you look at the methods of the `TDirect2DCanvas` class, it will seem there is very little on top of traditional GDI programming. However, if you look at the methods of the `ID2D1RenderTarget` interface, you'll see the huge changes. For example, drawing methods include `DrawBitmap`, `DrawLine`, `DrawRectangle`, `DrawRoundedRectangle`, `DrawEllipse`, `DrawGeometry`, `DrawText`, `DrawTextLayout`, and `DrawGlyphRun`. Most of these drawing methods output only the border of the corresponding shape or element, while a similar set of filling methods use the brush to color the surface of the element.

⁵⁰ Direct2D support for Vista is available as part of a platform update at <http://support.microsoft.com/kb/971644>. Notice also that there is no expectation to have this feature also available on Windows XP, as the underlying driver technology was much different.

The interface also supports transformations, layers, anti-aliasing, DPI settings, and status snapshots.

A lot of the power of Direct2D comes from the much more powerful graphic resources, including pens, brushes and fonts. As we'll see there is full support for gradients and other complex schemes. The Direct2D unit defines several new graphic objects, which don't inherit from their GDI counterparts or have common ancestors (as it happens for the Canvas). The Direct2D resources are managed by specific classes inheriting from `TDirect2DGraphicsObject`⁵¹, which often have constructors taking the corresponding GDI resource as parameter. Direct2D resource classes include `TDirect2DBrush`, `TDirect2DPen`, and `TDirect2DFont`.

Most other features, from colors to points, use new specific classes rather than the traditional ones, typically because their definition changes considerably. Points and all coordinates in Direct2D are based on floating point numbers, rather the integral ones. This is a huge difference. Again, there are helper functions for creating these new objects and conversions function for converting from their GDI counterparts. We'll see a few examples in the following demos.

Direct2D for a Form or a Canvas

It is now time to start looking at a first simple example of Direct2D, called `D2DIntro`. All you have to do to start experimenting with this technology is to create a Direct2D painting surface in the `OnPaint` event handler of a form:

```
procedure TD2DForm.FormPaint(Sender: TObject);
var
  d2dCanvas: TDirect2DCanvas;
  I: Integer;
begin
  d2dCanvas := TDirect2DCanvas.Create(Canvas, ClientRect);
  d2dCanvas.Brush.Assign(Canvas.Brush);
  d2dCanvas.BeginDraw;
  try
    d2dCanvas.Brush.Color := clRed;
    d2dCanvas.Pen.Color := clBlue;
    for I := 1 to 10 do
      begin
```

51 One of the advantages of using these Direct2D resources is that they are owned by their painting canvas and are disposed automatically when this is released. This makes it easier to manage these resources and much more difficult to incur in resource memory leaks (which are quite dangerous in the GDI world). On the other hand remember that when you create a graphic resource for a `Direct2DCanvas` you cannot use it in another one.

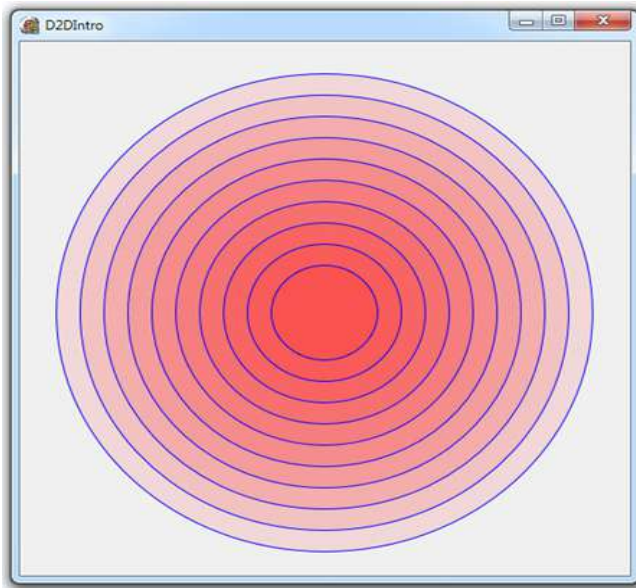
```

        d2dCanvas.Brush.Handle.SetOpacity(0.1 * I);
        d2dCanvas.Ellipse(10 + I * 20, 10 + I * 20,
            500 - I * 20, 500 - I * 20);
    end;
finally
    d2dCanvas.EndDraw;
    d2dCanvas.Free;
end;
end;

```

The `TDirect2DCanvas` object is created passing the GDI canvas as parameter, which is only one of the available options as I'll cover next. You can keep that object around or create a new one each time. What is important, though, is that all output operations take place between the `BeginDraw` and `EndDraw` calls⁵².

The program draws a series of concentric circles, using the `Ellipse` method, changing each time the opacity (or transparency) of the brush, which has a value between 0 and 1. This is one of the many features of the `ID2D1Brush` object (accessible with the `Handle` of the `TDirect2DBrush` object). Here you can see the visual effect, although the black and white version in the printed book makes the effect less clear than the full color version of the PDF.



⁵² This coding style allows all drawing commands to be completed “off-screen” and then displayed, to avoid “flicker” and other unwanted effects. For slow operations, though, you’ll see only the final result... after some delay.

This first demo program has a second feature. By selecting the local menu you can switch from painting on the GDI canvas to painting directly on the form. The menu items set the local `fUseCanvas` field, used to determine which of the constructors to call:

```

procedure TD2DForm.FormPaint(Sender: TObject);
var
    d2dCanvas: TDirect2DCanvas;
begin
    if fUseCanvas then
        d2dCanvas := TDirect2DCanvas.Create(
            Canvas, ClientRect)
    else
        d2dCanvas := TDirect2DCanvas.Create(Handle);
    d2dCanvas.BeginDraw;
    ... // as in previous listing

```

Depending on which of the `TDirect2DCanvas` you call, the VCL will initialize the underlying `ID2D1HwndRenderTarget` interface in different ways. If you pass the `Canvas` (or the `Handle` to the `Canvas` device context) and a rectangular area, Delphi will call the `CreateDCRenderTarget` function that lets you merge `Direct2D` and `GDI` calls over the same output surface. If you pass the form `Handle`, instead, the VCL ends up calling `CreateHwndRenderTarget`, which creates a specialized and optimized `Direct2D` output surface.

The notable difference, in the demo program, is that the native `Direct2D` surface will have a dark background, and the opaque brush will be merged with that, rather than the form color. You can change this by cleaning up the background by calling the `Clear` method of the underlying interface:

```

    d2dCanvas.RenderTarget.Clear(D2D1ColorF (clWhite));

```

Notice in this statement the conversion of a standard VCL color to its counterpart as a `D2D1ColorF` type.

Mixed Canvases

As I just mentioned, by creating a `Direct2D` surface tied to a `Canvas` (and its device context) you can merge output calls from both techniques while producing the output. This is also interesting as a way to figure out the differences in those output operations.

As an example consider the following painting code, which produces two circles on the left and right of the forms with two lines crossing over them (actually the `GDI` circle hides the `Direct2D` lines, but that's not the point of the demo as it depends on the specific brush being used):

```

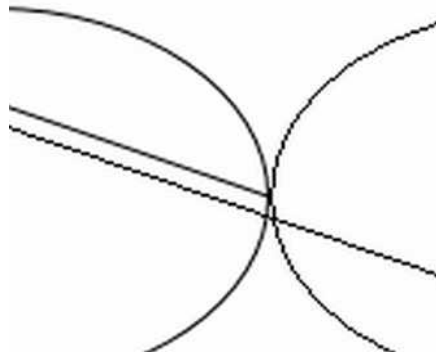
uses
  Direct2D, D2D1;

procedure TForm1.FormPaint (Sender: TObject);
var
  d2dCanvas: TDirect2DCanvas;
begin
  d2dCanvas := TDirect2DCanvas.Create (Canvas,
    Rect (0, 0, Width, Height));
  d2dCanvas.BeginDraw;
  try
    d2dCanvas.Brush.Color := clWhite;
    d2dCanvas.Pen.Color := clBlack;
    d2dCanvas.RenderTarget.Clear (D2D1ColorF (clWhite));
    d2dCanvas.Ellipse (100, 100, Width div 2, Height - 100);
    d2dCanvas.DrawLine (D2D1PointF (100, 100),
      D2D1PointF (Width - 100, Height - 100));
  finally
    d2dCanvas.EndDraw;
    d2dCanvas.Free;
  end;
  Canvas.Ellipse (Width div 2, 100, Width - 100, Height - 100);
  Canvas.MoveTo (110, 110);
  Canvas.LineTo (Width - 90, Height - 90);
end;

```

While you draw a circle in the same exact way, drawing lines with the two approaches is different as the Direct2D surface has a `DrawLine` method taking `D2D1PointF` parameters (which in turn use floating point coordinates). The canvas, instead, uses the classic `MoveTo` and `LineTo` calls.

The most significant difference, however, is in the output. By using anti-aliasing techniques, the Direct2D output is much smoother. Here is the central portion of the output image captured and zoomed to highlight the effect:



If it wasn't for the fact that this feature is currently available only in Windows 7 (with promises to port it back to Vista) I'd certainly switch to it if only for the

nicer and cleaner output effect. Another good reason, though is the extra features you gain, like the gradient brushes demonstrated next⁵³.

Gradients to the Max (With no Canvas)

There are many more features of Direct2D than I have time to demonstrate in this chapter. So I'll try to show a few more intriguing capabilities in the coming example, called D2DGradients. This program uses a different approach, as it creates and keeps around the Direct2DCanvas object, rather than creating a new one for each paint operation. In practice, the form has a private field declared as:

```
private
    d2dCanvas: TDirect2DCanvas;
```

This field gets initialized as the window itself is created, modified when the form is re-sized, and used by the painting code. Here is the creating and resizing code:

```
procedure TFormGradients.CreateWnd;
begin
    inherited;
    d2dCanvas := TDirect2DCanvas.Create(Handle);
end;

procedure TFormGradients.FormResize(Sender: TObject);
begin
    if Assigned(d2dCanvas) then
        (d2dCanvas.RenderTarget as ID2D1HwndRenderTarget).
            Resize(D2D1SizeU(ClientWidth, ClientHeight));
end;
```

The resize method is specific to the interface used if the render target is created from a windows handle. Notice that the program also checks for Direct2D support before starting, proving a specific error message:

```
procedure TFormGradients.FormCreate(Sender: TObject);
begin
    if not TDirect2DCanvas.Supported then
        raise Exception.Create('Direct2D not supported!');
end;
```

53 For a rather complete comparison of GDI and Direct2D, including optimization of each technique, you can see the DirectX developer blog post at:
<http://blogs.msdn.com/directx/archive/2009/09/29/comparing-direct2d-and-gdi.aspx>

Now that I've described the basic structure, let me focus on the actual painting code. The first thing the program does is paint the entire surface of the form with a gradient brush:

```

procedure TFormGradients.FormPaint(Sender: TObject);
var
    gradColors: array of TColor;
    Center: TD2D1PointF;
begin
    d2dCanvas.BeginDraw;
    try
        // define the gradient colors
        SetLength(gradColors, 4);
        gradColors[0] := clBlue;
        gradColors[1] := clAqua;
        gradColors[2] := clNavy;
        gradColors[3] := clFuchsia;

        // create the gradient brush, using colors and points
        Center := D2D1PointF(100, 100);
        d2dCanvas.Brush.Handle :=
            d2dCanvas.CreateBrush(gradColors, Center,
                D2D1PointF(300, 200), 900, 900);

        // paint the entire form with the gradient brush
        d2dCanvas.Rectangle(0, 0, ClientWidth + 50, ClientHeight + 50);
    
```

This prints the background with gradients ranging from various tones of blue to magenta⁵⁴. Next step is to write text painted with a slightly different gradient brush, based on the same colors. This new brush is assigned to the font, while the canvas brush is cleared to avoid seeing a rectangle around the text:

```

        // define a font with a gradient brush
        d2dCanvas.Font.Size := 60;
        d2dCanvas.Font.Brush.Handle :=
            d2dCanvas.CreateBrush(gradColors, Center,
                D2D1PointF(300, 300), 600, 600);
        d2dCanvas.Font.Style := [fsBold];
        d2dCanvas.Brush.Style := bsClear;

        // output some text with the current font
        strText := 'Delphi 2010';
        d2dCanvas.TextOut(200, 100, strText);
    
```

A gradient text over a gradient background starts looking interesting (and quite hard to accomplish in GDI terms), but there is more.

54 I won't try to show the output of this code here, as a gray-scale print won't be adequate. If you are interested you can have a look at my blog post:
http://blog.marcocantu.com/blog/direct2d_delphi2010.html

The program uses a transformation matrix for rotating the text while dimming its colors (by changing the opacity):

```
var
  matrix: TD2DMatrix3x2F;
begin
  ...
  for I := 1 to 10 do
    begin
      D2D1MakeRotateMatrix(I * 6, D2D1PointF(50, 50), @matrix);
      d2dCanvas.RenderTarget.SetTransform(matrix);
      d2dCanvas.Font.Brush.Handle.SetOpacity(1 - I * 0.1);
      d2dCanvas.TextOut(200, 100, strText);
    end;
```

There is a little more code to get the original transformation matrix and reassign it at the end, but the three code snippets just displayed sum up the code of the `OnPaint` event handler of this example.

For another example of Direct2D in Delphi, you can refer to the following post by Pawel Glowacki, who translated one of the Microsoft SDK demos (the Advanced Path Geometries Demo) in Delphi:

■ <http://blogs.embarcadero.com/pawelglowacki/2009/12/08/38861>

DirectWrite

We have seen that we can use the basic text drawing functions of Direct2D most like the GDI ones, but support for the precise display of text has much improved compared to the past. The new interfaces for managing font families, text layouts, text formats, and the like are collectively known as DirectWrite.

In Delphi you can access to the various DirectWrite objects using the factory object returned by the `DWriteFactory` global function of the `Direct2D` unit (which creates and initializes a singleton behind the scenes, implementing the `IDWriteFactory` interface). You can use this interface, defined in the usual `D2D1` unit, to create most of the core formatting objects. The list is extremely long and it would be somewhat pointless to show it here.

Instead, I'd rather focus on a simple example, which uses a `TDirect2DCanvas` and a few DirectWrite objects to draw a couple of strings on the form. The specific local variables used by the `FormPaint` method of the main form of the DirectWrite example are:

```
idwtFormat: IDWriteTextFormat;
idwtLayout: IDWriteTextLayout;
```



```
| matrix: DWRITE_MATRIX;
```

The first operation is the creation of a text format, obtained by passing the interface to be initialized to the `CreateTextFormat` method. The parameters are the font name, the (missing) font family, a series of constant display parameters, the size 35, a locale, and the object to be initialized:

```
| DWri teFactory.CreateTextFormat('Arial', nil,  
    DWRI TE_FONT_WEI GHT_LI GHT, DWRI TE_FONT_STYLE_NORMAL,  
    DWRI TE_FONT_STRETCH_SEMI _CONDENSED,  
    35, 'en-US', i dwtFormat);
```

The program passes this text format to the `DrawText` method of the `RenderTarget` low-level object, along with the text to display, its length, the output rectangle, and a brush:

```
| strText := 'Hello, Delphi';  
    d2dCanvas.RenderTarget.DrawText (PChar(strText),  
        Length (strText), i dwtFormat,  
        Rect (10, 10, 500, 200), d2dCanvas.Brush.Handle);
```

In the code above I pass a `TRect` to the `DrawText` function where it would expect a `D2D1_RECT_F` structure. This is possible because the latter data structure has an `Implicit` conversion operator⁵⁵ defined as:

```
| class operator Implicit(AValue: TRect): D2D_RECT_F;
```

There are similar conversion operators for other rectangle data structures and a couple of point data structures of the `D2D1` unit.

The second output operation done by the program in the `FormPaint` event handler uses the same technique, but with a bold, oblique, and expanded text format. Two new calls align the text to the center and set very tight line spacing (the value 28):

```
| strText := 'This is a DirectWrite oblique example';  
    DWri teFactory.CreateTextFormat('Arial', nil,  
        DWRI TE_FONT_WEI GHT_EXTRA_BOLD, DWRI TE_FONT_STYLE_OBLI QUE,  
        DWRI TE_FONT_STRETCH_EXTRA_EXPANDED, 35, 'en-US', i dwtFormat);  
  
    i dwtFormat.SetTextAl ignment(DWRI TE_TEXT_ALI GNMENT_CENTER);  
    i dwtFormat.SetLi neSpaci ng(  
        DWRI TE_LI NE_SPACI NG_METHOD_UNI FORM, 28, 0);  
  
    d2dCanvas.RenderTarget.DrawText (PChar(strText),  
        Length (strText), i dwtFormat,  
        Rect (10, 110, 500, 200), d2dCanvas.Brush.Handle);
```

55 For more details on operators overloading in general and conversion operators in particular you can refer to my *Delphi 2007 Handbook*.

The static output of the form should look like the following image:



Using the Windows Imaging Component

Another new graphic-related feature of the VCL is its support for the Windows Imaging Component (WIC). This is a Microsoft framework for working with images and their metadata, which supports several image formats and can be extended with new image formats by software and hardware vendors (like digital camera makers). Not only can the WIC can display images, but it has also a lot of image processing capabilities built into it and independent from the actual image format.

The WIC is available in Windows 7 and in Vista, but it is also possible to install it on Windows XP⁵⁶, making it available on a broader base than some of the technologies discussed earlier in this chapter. By default, WIC has the following built-in image formats (here listed with the corresponding mime types):

- BMP (Windows Bitmap Format), BMP Specification v5
- GIF (Graphics Interchange Format 89a), GIF Specification 89a/89m
- ICO (Icon Format)
- JPEG (Joint Photographic Experts Group), JFIF Specification 1.02
- PNG (Portable Network Graphics), PNG Specification 1.2
- TIFF (Tagged Image File Format), TIFF Specification 6.0
- Windows Media Photo, HD Photo Specification 1.0

56 For the installation of the Windows Imaging Component in Windows XP you can see corresponding page of the Microsoft download site at <http://www.microsoft.com/downloads/details.aspx?FamilyID=8e011506-6307-445b-b950-215def45ddd8> or type in your browser the shortened version <http://bit.ly/aQn45N>

Delphi 2010 support for the Windows Imaging Component is based on the new `TWImage` class, a `TGraphic` descendant defined in the `Graphics` unit like the classic `TMetafile` and `TBitmap` VCL classes. The image formats handled by this component are all those mentioned earlier, as you can see in the enumeration underlying the `ImageFormat` property:

```
TWImageFormat = (wiFBmp, wiFPng, wiFJpeg,
    wiFGif, wiFTiff, wiFWMPPhoto, wiFOther);
```

However, the `Image` component registers only the `TIFF` graphic format (or to be more precise, file extensions) for use with the `TWImage` class, as you can see from the following VCL code snippet:

```
constructor TFileFormatsList.Create;
begin
    inherited Create;
    Add('wmf', SVMetafiles, 0, TMetafile);
    Add('emf', SVEnhMetafiles, 0, TMetafile);
    Add('ico', SVIcons, 0, TIcon);
    Add('tiff', SVTIFFImages, 0, TWImage);
    Add('tif', SVTIFFImages, 0, TWImage);
    Add('bmp', SVBitmaps, 0, TBitmap);
end;
```

Including specific units, like the `JPEG` unit, the VCL registers more formats:

```
initialization
    TPicture.RegisterFileFormat('jpeg', sJPEGImageFile, TJPEGImage);
    TPicture.RegisterFileFormat('jpg', sJPEGImageFile, TJPEGImage);
```

This means that in Delphi 2010, when running on a version of Windows with `WIC` support, you can manage the `JPG` files using this last component. All you have to do is to declare the following:

```
TPicture.RegisterFileFormat('jpg', 'JPEG Image', TWImage);
```

This is code you'll find in the `TiffViewer` example, which lets you open in an `Image` component a `TIFF` or a `JPEG` file in an `Image` component using the `TWImage` class. You don't have to do anything special to gain this support, as all you have to do is to open the file (`Image1.Picture.LoadFromFile`) and the `Image` component will use the support class registered for the file format.

WIC Transformations

Again, there is very little you have to do to use `TIFF` support... just open a file in the `Image` component. However, it is important to notice that there are two sides to the `WIC`. One is loading and saving files with given formats. The

second side is the ability to process the images. To accomplish this you have to access to the `IWICImageFactory` interface, available through the `IImageFactory` property of the `TWICImage` class. This COM factory lets you create specific COM objects that expose given interfaces:

- the `IWICFormatConverter` interface to converter formats
- the `IWICBitmapScaler` interfaces for the scaling bitmap
- the `IWICBitmapClipper` interfaces for bitmap clipping support
- the `IWICBitmapFlipRotator` interfaces to flip and rotate images
- the `IWICColorTransform` and `IWICColorContext` interfaces that can be used for color management

Again, there is a plethora of methods and options, much more than I can cover in this section. What I'll do, instead, is to show you a practical example of how a Delphi application can manipulate images. The first part of the code extracts information from the image currently loaded in the `TWICImage` object, using the `IWICBitmap` interface:

```
var
  wi cI mg: TWI CImage;
  i BmpSource: IWICBitmapSource;
  pui Wi dth, pui Hei ght: UI NT;
begin
  if Image1.Picture.Graphic is TWICImage then
  begin
    wi cI mg := TWICImage (Image1.Picture.Graphic);
    i BmpSource := wi cI mg.Handle as IWICBitmapSource;
    i BmpSource.GetPixelFormat(pPixelFormat);
    i BmpSource.GetSize(pui Wi dth, pui Hei ght);
    Log ('Original: ' + IntToStr (pui Wi dth)
        + '/' + IntToStr (pui Hei ght));
```

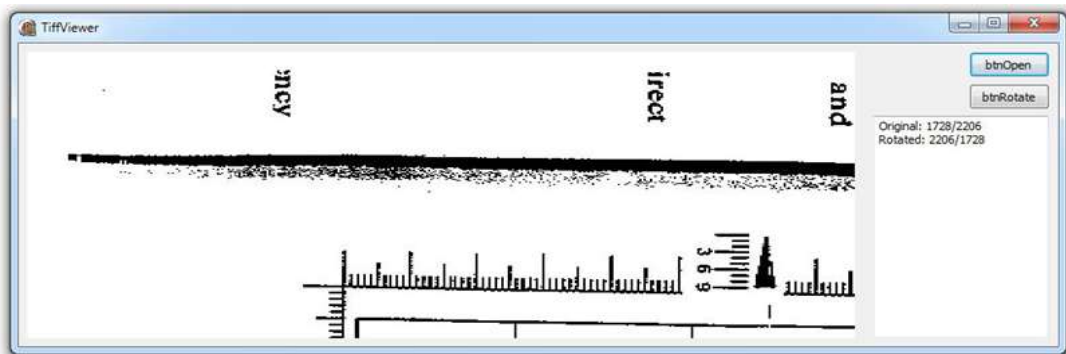
This outputs the width and height of the bitmap. The second step is to extract the flip/rotator interface, assign it to the source bitmap, ask for a given operation (in this case a 90 degree rotation), and again display the width and height of the rotated image:

```
var
  i FI pRot: IWICBitmapFlipRotator;
  i BmpSource: IWICBitmapSource;
  pui Wi dth2, pui Hei ght2: UI NT;
begin
  ...
  wi cI mg.ImageFactory.CreateBitmapFlipRotator(i FI pRot);
  i FI pRot.Initialize (i BmpSource, WI CBitmapTransformRotate90);
  i FI pRot.GetSize(pui Wi dth2, pui Hei ght2);
  Log ('Rotated: ' + IntToStr (pui Wi dth2) + '/' +
      IntToStr (pui Hei ght2));
```

Finally we have to extract the rotated bitmap (passing the correct dimensions for the rectangle of the source image) and assign it as the source of the TWI C I mage object:

```
var
  wi cBi tmap: I WI C Bi tmap;
begin
  wi cI mg. I magingFactory. CreateBi tmapFromSourceRect(
    i Fl i pRot, 0, 0, pui Wi dth2, pui Hei ght2, wi cBi tmap);
  i f Assigned (wi cBi tmap) then
    wi cI mg. Handl e := wi cBi tmap;
  I mage1. Repai nt;
```

You can see a sample result in the image below. Consider that you can do similar transformation for each file format, but need to load them in the TWI C I mage object first. In this case I've loaded and rotated a sample TIF image (the test page of a FAX), which is included along with the source code of the example:



Other New VCL Features

Not all new VCL features are advances at the operating system and SDK level related with Windows 7. The library has seen countless small improvements. I'm going to focus on them in the remaining portion of this chapter. Let me start with some noteworthy but simpler changes (this is not an exhaustive list, as I picked what I felt most relevant), while I'll later focus on improved property editors and changes in multi-language support. Here is a list of these notable changes:

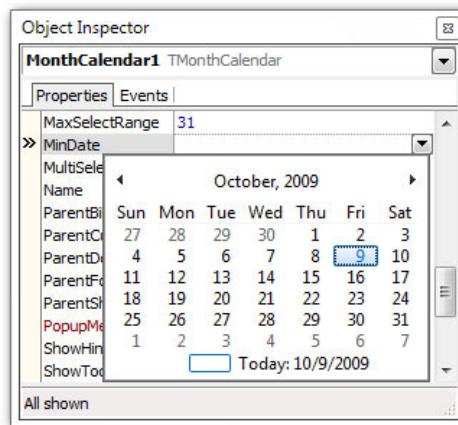
- All grids (DrawGrid, StringGrid, DBGrid, and also the grid-descendant ValueEditor control) now support themes. See Chapter 7 for an actual

example featuring a DBGrid. Grids and the ValueEditor also support gradients and other graphical enhancements.

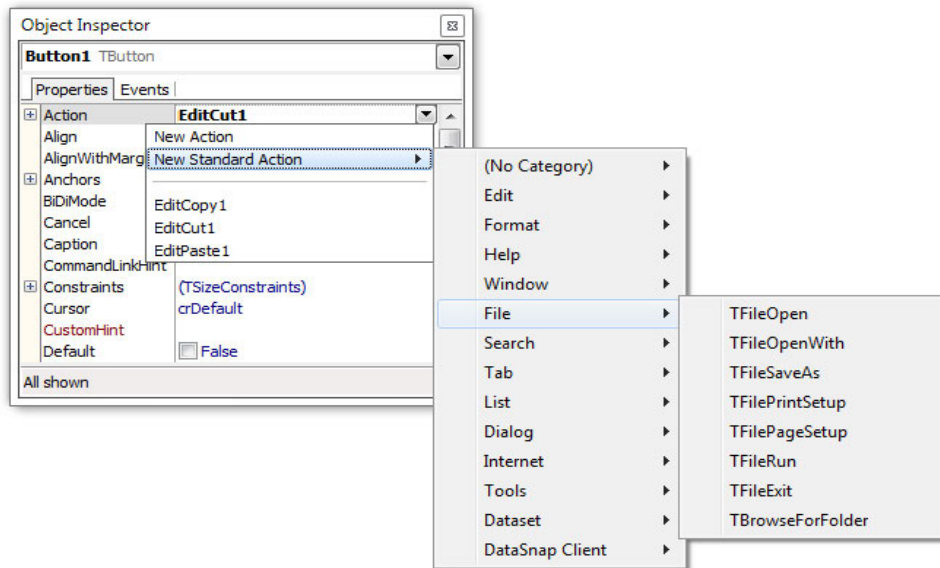
- The ProgressBar components now supports 32-bit values for the properties Position, Min, and Max. Older Windows platforms won't support them, though.
- The Assign method of a TBitmap image can take also a TIcon parameter, letting you convert an icon into bitmap.
- The two buttons (of class TEditButton) on the sides of the ButtonedEdit component (introduced in Delphi 2009) now support individual hints.
- When using the CategoryButtons control (the one used in the Delphi IDE for the Tools Palette) you can enable in-place editing for the categories. The component has seen also several other improvement and fixes.
- The CheckListBox component has a new CheckAll method.
- The SpeedButton control can now be properly painted over a Glass surface.
- The global Screen object has a new CaptionFont property, that works together the MessageFont property, introduced in Delphi 2009 and used to determine the font of some of the message dialogs.

Property Editors for Actions and Dates

Beside the changes in VCL components, Delphi 2010 has a couple of very interesting additions in terms of changes to property editors, which spawn across multiple components. Every time you have a TDate property, you can now use a MonthCalendar to pick a value at design time:



A much more useful update was made to the editor of the Action property. In the past you could only pick one of the available actions from a drop down list, now you can also define a new (custom) action or a new standard action in one of the available categories:



This feature⁵⁷ is available only if the current form (or current designer) hosts or is connected with an ActionList or ActionManager component.

Input Language and Language Libraries

In Delphi 2010 there have been a couple of significant changes for developers who build international applications. The first is that the Windows `WM_INPUTLANGCHANGE` message now triggers an internal component message, `CM_INPUTLANGCHANGE`, which is broadcasted to all windowed controls. You can intercept and process this message in a `TWinControl` descendant, by adding a message handler method like:

```
procedure CM_INPUTLANGCHANGE(var Message: TMessage);
message CM_INPUTLANGCHANGE;
```

⁵⁷ The new Action property editor was first mentioned on his blog by Chris Bensen at: <http://chrisbensen.blogspot.com/2009/08/rad-studio-2010-actions.html>

The second change relates to the way the VCL loads localized resource files in the `LoadResourceModule` function of the `System` unit. This is used, in particular, with resource files generated by the Internal Translation Manager (activated with the `Project | Languages` menu). In the past Delphi would use either a local override in the system registry or use the current locale specified in the Regional Settings of the Control Panel. In Delphi 2010, instead, the VCL still looks for the local override in the system registry, but, in case this is not set, it uses the default user UI language (unless the program is running on an old version of Windows, in which case the classic behavior is respected).

The default user UI language is the active language⁵⁸ of the operating system, determined in the VCL by calling the `GetUserDefaultUILanguage` API. In other words, by changing the Regional Settings of the Control Panel you will no longer affect the resources being loaded for your application.

We could discuss which strategy is more flexible as both have merit, but my point here is that this is an important change that is not mentioned in the Delphi documentation and could easily get overlooked by developers. The `LoadResourceModule` function of the VCL, responsible for this behavior, has been completely rewritten in Delphi 2010, with a different effect depending on which version of Windows under which you are running your application. I won't delve into the Translation Editor and its behavior here, nor delve into the detailed effects of this change, but only warn you about the existence of this change⁵⁹.

Minor Incompatibilities with “Growing” Enumerations

The last thing I want to mention about the VCL is a very subtle change that might prevent some existing programs from compiling. Some of the enumerations used by the VCL, which have not been modified in a long time, have been extended with new values.

58 All recent version of Windows can be installed with a multilingual support that lets you switch the operating system language at run time, rather than reinstalling a new language-specific version of the operating system.

59 Thanks to Jaakko Salmenius of Sisulizer for bringing this to attention in a Delphi forum.

One of them is `TShiftState`, which now supports touch and pen operations and is defined as:

```
TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft,
    ssRight, ssMiddle, ssDouble, ssTouch, ssPen);
```

Another enumeration that has a new extra value is `TControlState`. The problem arises as you convert one of the enumerated values to its numeric counterpart using a direct cast (rather than the `Ord` function). In Delphi 2009 a `TControlState` value is stored in a `Word`, in Delphi 2010 in a `Cardinal`. So, following this example, the classic Delphi code below doesn't compile in Delphi 2010:

```
var
    w: Word;
begin
    w := Word (ControlState)
```

You can easily update it to:

```
var
    c: Cardinal;
begin
    c := Cardinal (ControlState);
```

These changes have an extremely limited effect, as in general it would be recommended to use the `Ord` function and store the value in an `Integer`. However there are many custom controls that use such a low-level coding style, and those might get affected and fail with a compiler error. As I've bumped into two such situations, there are certainly few around, so I felt worth mentioning the issue... since when you realize the problem the fix becomes almost trivial.

What's Next

In this chapter I've delved into new features of Windows 7, changes to the VCL to address those new capabilities, and other general change to the Delphi component library. One of the most relevant new features of the VCL, partially related with Windows 7, is the support for touch and gestures, which is covered in the next chapter. After that I'll move to changes and new features to the database part of the VCL.



www.DelphiDeveloperDays.com
Featuring Marco Cantù and Cary Jensen

Marco Cantù and Cary Jensen jointly offer Delphi Developer Days, multi-day live events with both joint sessions, presented by Marco and Cary together, as well as simultaneous tracks, where Cary and Marco break out into separate rooms to present individual topics.

All attendees receive very detailed course books, which cover all topics presented by Marco and Cary plus all of the source code. Attendance is limited to keep the event intimate so that you'll have the opportunity to ask them about any of the topics that they covered, as well explore other Delphi questions that you might have. Whether you are using the latest version of Delphi, or are developing with an older version, you will come away with loads of information that will improve your development and make you more productive.

Delphi Developer Days Tour Schedule for 2010

- Washington DC/BWI Airport Area: May 11-12, 2010
- Chicago Area: May 14-15, 2010
- Los Angeles Area: May 17-18, 2010
- London, UK: May 26-27, 2010
- Frankfurt, Germany: May 31 - June 1, 2010

Future Dates for Delphi Developer Days

Marco and Cary plan to add new tour dates in the future. To be notified of additional dates, visit <http://www.DelphiDeveloperDays.com/NotifyMe.html>.

www.DelphiDeveloperDays.com brought to you by:



Chapter 6: Touch And Gestures

With mice and keyboards being part of the computers landscape since the Nineties, experimental user interfaces have been focused mostly on script recognition (starting with the Apple Newton, the Graffiti input system of Palm hand computers, and including tablet PC support in Windows) and voice recognition. These experiments have never reached a critical mass or affected everyday computer users.

Over the last couple of years, however, a slow revolution has started outside of the PC realm. The two most prominent and known examples are the Wii controllers (which receive input from three-dimensional spatial movement) and the touch support of the iPhone (which lets you move a finger across the screen to issue a command).

Many people are now expecting that this user interface revolution will soon hit desktop PCs. Among the companies backing this idea there is Microsoft, which added extensive support for touch in Windows 7, and Embarcadero Technologies, which is pushing touch and gesture support in Delphi 2010.

There are many other elements beyond touch, including for example the Sensor and Location API that is also part of Windows 7 (which you can use in Delphi

by directly interfacing to it). In this chapter, though, I'll cover only touch and gestures, focusing specifically on gestures as touch support requires specific hardware that is still hardly popular. I'll also cover related VCL features, like the new TouchKeyboard control.

From Single Touch to Multi-Touch

Touch screens have been around for several years now and have been popular in kiosk applications, vertical-market tasks like restaurant ordering systems, and to a more limited extent in tablet PCs. The classic touch screen can intercept the position of your finger on the surface, but returns only one given coordinate. This makes it quite a rough approximation of the position as your finger, unlike a dedicated pen with a more fine tip, will generally touch multiple screen pixels at a time⁶⁰.

In technical terms, a classic touch screen behaves like a mouse sending mouse down and mouse up Windows messages to the operating system (and hence to the application). This is what is now generally called a single-touch system.

Newer touch screens, in fact, can intercept multiple pressure points at the same time, and send them all in parallel to the system. These multi-touch⁶¹ systems bring two different advantages.

First, if you press your big finger over the screen, the driver will reduce the points in proximity to a single one but with a better resolution, so it would be easier to figure out your intended operation. Also, any movement you make with your finger on the screen is interpreted in more precise way, making it easier to perform gestures.

Second, you can select two or more on-screen elements at the same time, by using two or more fingers. There are currently systems being sold which are

60 If you've ever tried using the small buttons of a Windows CE device with your fingers instead of the specific pen you probably know what I mean. I consider it a dreadful experience, but having to use the pen to select a contact and make a phone call while you are walking is even less natural than trying to hit the name with your finger, or even your fingernail to be more precise!

61 As odd as it might sound, the term "multi-touch" is actually a trademark of Apple Inc. as listed on <http://www.apple.com/legal/trademark/appletmlist.html>.

capable of intercepting up to ten touch positions (and although it would be quite odd for a single user to need more, multi-user systems like Microsoft Surface can benefit from more than ten touch points). You can also move two or more fingers on the screen to manipulate the objects you are touching. A classic example is stretching a picture by dragging away the opposite borders with two fingers.

For multi-touch you need support both at the operating system level and at the hardware level. At the operating system level, Windows 7 introduces specific system messages, like `WM_TOUCH`. This message carries a wealth of information about the input activities of the user, computes the touch to single coordinates plus a couple of key shift states of the traditional mouse messages.

At the hardware level, you need multi-touch hardware with drivers specifically tailored for Windows 7 (as older touch support drivers will simply mimic the traditional mouse messages, as in single touch devices).

Another way to think about touch versus the mouse (and its derivatives) is to consider that in the first case the system works with absolute positions, while in the second it is relative movements that matter. When you touch a screen (or optionally a touch pad with full touch support), you are indicating a specific location. With the mouse (and some movement oriented touch pads) you move it from the current position to a new one, but it is the relative movement that matters. In fact, if you lift the mouse and lower it down in a different place, the effect is like it didn't move!

Touch Hardware

You might think that touch hardware comes in the form of touch screens you add to your existing PC. This is hardly the case, although some multi-touch monitors are starting to appear. In most current offers, touch screens are part of a single computing device, either a touch screen laptop computer or an all-in-one desktop (or wall-mounted) device.

Here I won't provide a complete overview of touch-enabled hardware, but focus only on Windows 7 enabled devices⁶². These includes portable computers (not-

⁶² You can find a very comprehensive history and board review of available touch hardware, beyond Windows, in an article at <http://medlibrary.org/medwiki/Multi-touch>.

ably from Dell and HP), all-in-one devices (mostly from HP), and computers with multi-touch touch pads (like those from Acer and Dell):

- A typical touch-enabled laptop (with a rather small screen is) HP's TouchSmart tx2z and Dell's Latitude XT2, but also machines from Lenovo and Acer.
- All in one systems from HP include HP TouchSmart 300 and 600, Dell's Inspiron One 19 Touch, and Sony's Vaio L.
- If you are looking for an external touch monitor to be attached to an existing PC, you can refer to the Dell's SX2210T 21.5"W Multi-Touch Monitor.
- If you are looking for a simpler solution, consider multi-touch internal touch pads, like the one in Dell's Inspiron Mini 10. I tried it and it looks interesting. Many recent laptop computers offer a similar solution.

Multi-Touch Pads

Speaking of multi-touch pads, I'm convinced they should probably be favored above touch screens, not just because of the reduced cost. Adding a large multi-touch touch pad to an existing computer would be a very good solution for moving to multi-touch. A touch pad works better than a touch screen for standard PC users, as touching the screen is very ineffective both in terms of arm fatigue and in terms of covering what you are looking at with your hand.

I noticed, in particular, the nice looking Bamboo Touch device from Wacom (and even bought one), but this device comes up very short in terms of Windows 7 support, as its driver maps multi-touch interaction to traditional mouse operations and the driver uses relative movements rather than absolute positions. It is not that the device won't be capable, but the driver falls quite short. Actually Microsoft itself is not pushing this class of hardware, favoring touch screens, but I'd argue this is OK for special purpose PCs, but not for mainstream office users.

The Theory Behind Gestures

As I mentioned in the introduction of the chapter, the main topic is touch support but with the specific focus on gestures.

But what is a gesture? It is basically a movement of the input device following a specific path and within a given time. The path of a gesture is not absolute in its

extension, but is represented by a vector of points which can generally be scaled at will. This means you can make a more ample or compact gesture and they'll still both be recognized. In other words, a gesture is represented by a sequence of points and some further parameters that are passed to an engine capable to recognizing a gesture against a given gestures catalog.

What is very relevant to notice up front is that I intentionally used the vague term of “input device”, as gestures can be issued not only by moving a finger over a touch screen but also by moving the mouse over a standard flat surface.

In other words, while multi-touch support requires specific hardware and operating system support, gestures support can work on any version of Windows and with any input device (except a keyboard, of course, but including the single touch pads that most laptops have these days).

Towards a Touch-Based UI

Supporting touch-based interfaces is not just a matter of using larger buttons and virtual touch keyboards, but often requires a significant redesign of the program user interface. Every input operation should account for more imprecise input, keyboard actions should be minimized (using auto complete features like many cellphones do, for example), and you should move away from the traditional user interface whenever required.

A very good example of an innovative approach is the use of gestures, as I covered in the first part of the chapter. When you have a touch screen, performing a gesture will generally be much easier than selecting a toolbar button. However, for regular PC users, pressing a shortcut key will remain faster than both the mouse and the touch operation.

In other words, while in the past we had to balance the needs to mouse users and keyboard users, now we have a third category (touch or interactive users) to take into account in general purpose applications. Of course, if the program is not for the PC but for a dedicated computer, it might not have a mouse or a keyboard and this solves part of the problem. But with the introduction of touch screen in regular PCs, we'll have to face the problem of different users having different input preferences when based on the same hardware.

The Gesture Manager of the VCL

In Delphi 2010 the VCL adds to its core features a gesture management architecture. This is built into the TControl hierarchy, with the new OnGesture event surfacing in all visual components. However, the actual gesture management code can be selectively compiled into the VCL or not, thus avoiding the extra overhead⁶³ in case you are not interested in using gestures support.

The core class of this architecture is the new GestureManager component. It can be used to handle gestures performed with touch hardware, pen and tablet input or even traditional mouse input. The GestureManager defines a set of standard and predefined gestures, but you can add custom gestures directly at the application level or by installing packages that register gestures.

The GestureManager also handles special “interactive” gestures that require touch hardware. These special gestures include panning, zooming, rotating, two finger taps, and press and tap operations.

As you enable gestures for a given control (by hooking the GestureManager component to it), any gesture performed over the control⁶⁴ will fire the OnGesture event of the control, unless there is a specific action tied to it.

A Basic Gesture Example

Rather than describing the complete VCL gesturing architecture in theory, let us start by building a very simple first example I've called Gestures01. The program has a form with a panel and a memo control, plus the GestureManager component. This last component doesn't have specific setting, as its editor let's you add custom gestures to it, something we will focus on later.

The management of gestures takes place in the Touch property of the target control, for example the form. This property is slightly a misnomer, as it handles all touch and gesture related operations, through its sub-properties. If

63 The gestures overhead is about 300KB to 400KB in executable size, which is why it was worth taking additional measures to avoid it when not needed.

64 Performing a gesture over the control means making the proper mouse or finger movement starting over the control. If you move over other controls while performing the gestures, it won't matter. What is relevant is the starting position.

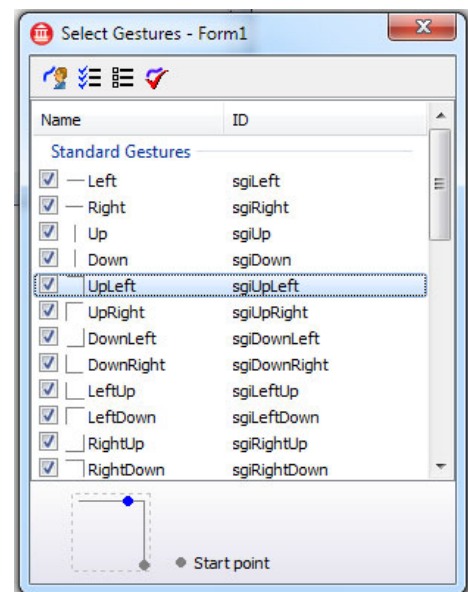
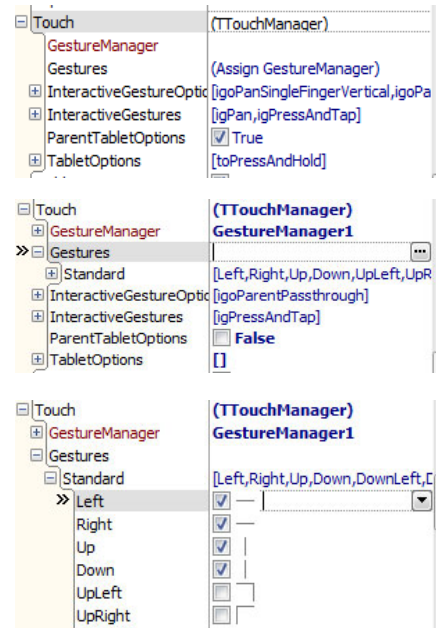
you expand the Touch property (which is of type `TTouchManager`) for a control, you should see something similar to the image on the right, with the Gestures sub-property disabled as the `GestureManager` sub-property is not assigned.

As soon as you connect a `GestureManager` control, the interface of the Object Inspector for the property changes considerably, with the collections of available gestures (by default the Standard ones) listed under the Gestures property, as you can see in the second image here on the right.

In the Object Inspector you can also expand the Standard gestures node, and enable or disable individual gestures for the control, as depicted in the third image of the sequence.

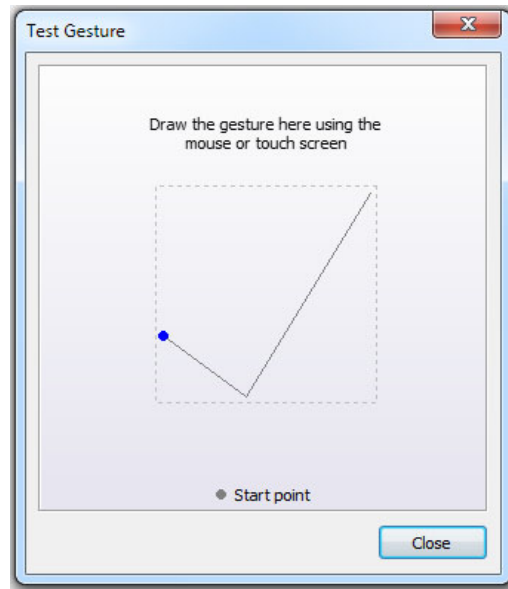
Each element of this collection of gestures (a `TGestureCollectionItem` element) has a name, a graphical representation, and (optionally) the action to which it is tied. We'll get to using actions for gestures in a second example. For now, we can just enable all of the gestures for the form (and the panel). This is done more easily in the gestures collections editor rather than in the Object Inspector, as you have buttons to select and deselect all standard actions. The gestures collections editor is visible here on the right.

In this editor you can see the gesture name and graphical description, its internal identifier (a constant) and a graphical animated preview of the gesture itself, with the start point in gray and the movement highlighted by a blue



170 - Chapter 6: Touch and Gestures

point. You can see an enlarged version of this preview in a gesture test window by double clicking on one of the entries:



Also notice that as the editor caption points out, you are picking the gestures for a given target control (in this case Form1), not at the GestureManager component level. The manager has all possible gestures for the entire application, each form or control picks only those it is capable of handling.

Now, having selected all of the standard gestures for the form and the panel, we can write some code for their generic OnGesture event handler. For the panel I simply log the fact the event was intercepted to the memo, nothing more:

```
procedure TForm1.Panel1Gesture(Sender: TObject;  
  const EventInfo: TGestureEventInfo; var Handled: Boolean);  
begin  
  Log ('Panel Gesture');  
end;
```

I added this event handler to let you figure out how gestures are associated with the controls underneath, and what matters most is the starting position of the gesture as I mentioned in an earlier note.

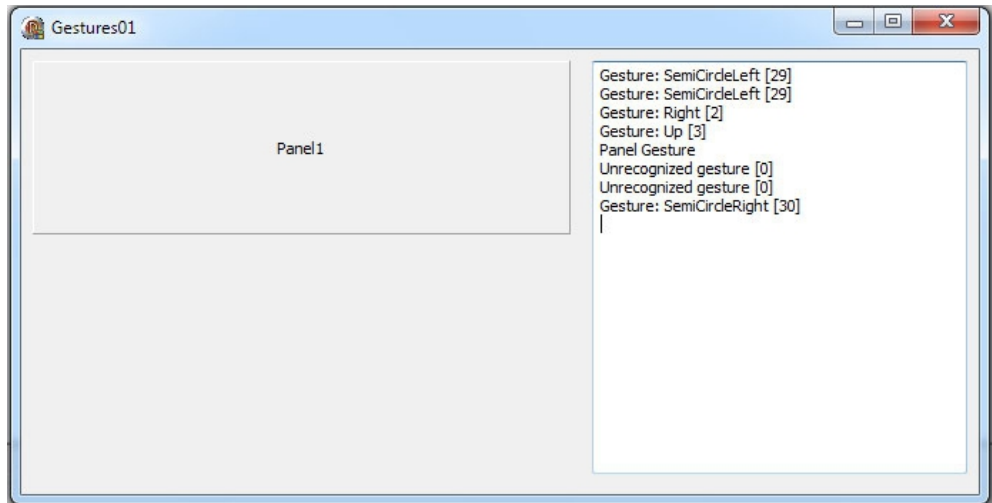
For gestures performed on the form, instead, I want to list some of the gesture information, notably their name and ID. The name is extracted from the collection of the corresponding component, which might not exist in case there is no match. Here is the code:

```

procedure TForm1.FormGesture(Sender: TObject;
  const EventInfo: TGestureEventInfo; var Handled: Boolean);
var
  nGesture: Integer;
  gestureItem: TCustomGestureCollectionItem;
begin
  nGesture := EventInfo.GestureID;
  gestureItem := GestureManager1.FindGesture(sel f, nGesture);
  if Assigned (gestureItem) then
    Log (Format ('Gesture: %s [%d]', [gestureItem.Name, nGesture]))
  else
    Log (Format ('Unrecognized gesture [%d]', [nGesture]));
    Handled := True;
end;

```

The output of the program is a list of gestures performed on the form, with a simple indication for those originating on the panel:



The Standard Gestures

In the previous example we have enabled all of the standard gestures to the form and the panel, but so far I haven't given you any indication of which of these are predefined gestures. You can find a complete list with a graphical representation of each gesture in the help page:

|| [help: //embarcadero.rs2010/rad/TStandardGesture_Enum.html](http://embarcadero.rs2010/rad/TStandardGesture_Enum.html)

Here I've listed all of the standard gestures logically grouped (the grouping is mine), providing only a few of those images:

- Gestures with a single movement: Left, Right, Up, and Down.

- Gestures with two perpendicular movements: UpLeft, UpRight, DownLeft, DownRight, LeftUp, LeftDown, RightUp, and RightDown. There are also versions requiring one side to be twice as long of the other: UpLeftLong, UpRightLong, DownLeftLong, and DownRightLong (depicted here on the side).
- Gestures requiring two opposite movements: UpDown, DownUp, LeftRight, RightLeft and a repeated left and right movement, Scratchout.
- Gestures mapped to geometrical shapes: Triangle, Square, Circle, and DoubleCircle.
- Gestures with special circular movements: SemiCircleLeft, SemiCircleRight, Curlicue, DoubleCurlicue (shown here).
- Gestures corresponding to two perpendicular diagonal movements: ChevronUp, ChevronDown, ChevronLeft, and ChevronRight; plus one in which the upwards movement is twice as long than the downwards one, Check (shown earlier in the Test Gesture pane).



Gestures and Actions

Although you could use the `OnGesture` event handler to figure out which gesture the user performed and connect a specific operation, the gesture architecture of the VCL does well at tying gestures to actions. The idea is that you can issue the same command using a menu item, pressing a button, pressing a shortcut key, or performing a gesture, and the simplest way to achieve this is to hook the menu item, button, shortcut, and gesture to a single action.

I think this approach adds to the already powerful `ActionManager` architectures of Delphi, and its simplified `ActionList` sibling, making it even more the natural core element of most Delphi GUIs.

In this example, called `EditGestures`, I have added to the main form an `ActionList` component with a couple of standard edit actions, plus a custom one:

```
object ActionList1: TActionList
  object EditCut1: TEditCut
    Category = 'Edit'
    Caption = 'Cu&t'
  end
  object EditPaste1: TEditPaste
    Category = 'Edit'
    Caption = '&Paste'
  end
end
```

```

object EditClear1: TAction
  Category = 'Edit'
  Caption = 'EditClear1'
  OnExecute = EditClear1Execute
end
end

```

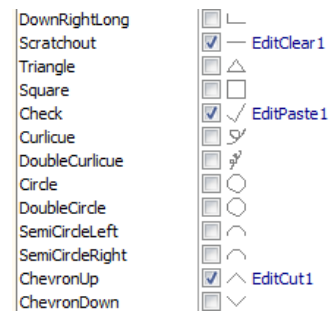
The OnExecute method of the custom action clears the edit box content:

```

procedure TFormEditGest.EditClear1Execute(Sender: TObject);
begin
  if (ActiveControl is TCustomEdit) then
    TCustomEdit(ActiveControl).Clear;
  end;

```

Now while these actions can be activated using the shortcut commands, the program has no menu to execute them, but the form adds gesture support, connecting three gestures to the actions, as you can see in the image of the Object Inspector here on the right.



To have a clearer view of the actions connected to the active gestures for a control, what you can do is copy the properties of the GestureManager component, which will list the gestures binding for each control connected to it (in this case only the form, the owner)⁶⁵. For each active gesture you can see the corresponding action:

```

object GestureManager1: TGestureManager
  GestureData = <
    item
      Control = Owner
      Collection = <
        item
          Action = EditPaste1
          GestureID = sgiCheck
        end
        item
          Action = EditCut1
          GestureID = sgiChevronUp
        end
      end
    end
  end

```

⁶⁵ Notice that gesture selection is defined for each control but not stored with the control, but in the centralized GestureManager. This happens mostly because the controls can pull-in gesture data from the GestureManager, but are unaware of the gestures-related code. Were the controls aware of gestures, any application would have to link the gesture related code. As the design decision was to keep gesture support outside of compiled applications that don't use it, this idea of keeping gesture data outside of the control was a necessary consequence.

```

        item
            Action = EditClear1
            GestureID = sgiScratchout
        end>
    end>
end

```

As you can see I've hooked the Paste action with the Check gesture, the Cut action with the ChevronUp (^), and the custom Clear action with the Scratchout gesture. Now, there is very little I can show you about the program in an image, so you'll have to test it for yourself⁶⁶.

There is one more thing the program does, it logs gesture operations and action requests, to help you better understand the flow of events. The program handles the OnGesture event of the form, which will be triggered only for gestures that are not recognized or not managed. That is, if the gesture has a corresponding action, the OnGesture event will not fire. The second hook is in the OnExecute event of the action list, which simply logs the action name. This is invariably called before performing the actual action.

Custom Gestures

Although the list of predefined gestures is quite rich, there might be specific mouse movements that make sense for your application and for specific actions you want users to perform. In this cases, you can add custom gestures to the GestureManager component.

Even if you don't plan on adding custom gestures to your application, it is worth looking at the details of this process, as it explains you what “matching a gesture” exactly means and which parameters you can fine-tune to improve matching.

While doing so, we'll also take advantage of some gestures-related components:

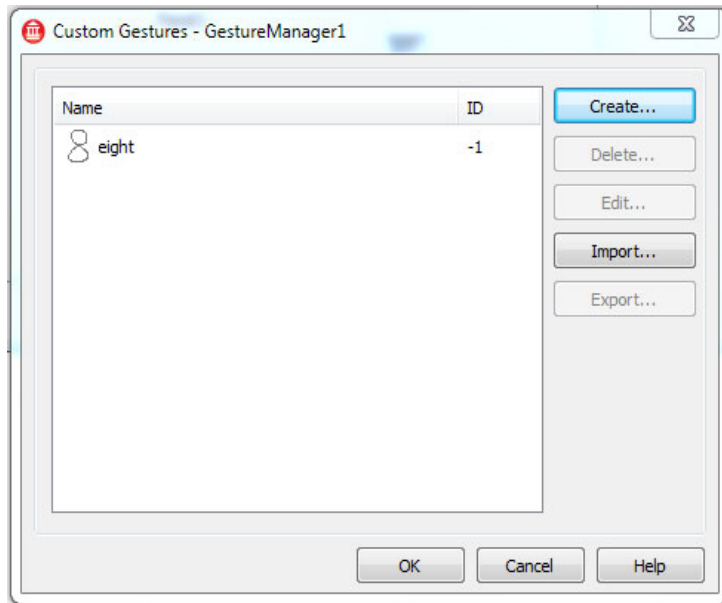
- The GestureListView can be used to show a summary of the active gestures.
- The GesturePreview lets you show and test a gesture.
- The GestureRecorder lets your users define their own custom gestures (this one I won't demonstrate, as I consider it of limited practical use⁶⁷).

⁶⁶ Or see the project video, listed at <http://www.marcocantu.com/dh2010/videos.htm>

⁶⁷ An interesting case for custom gestures is to use it as a security measure: ask a user to make a gestures and then use that to recognize the user instead of asking for a password.

Let me start with the definition of a custom gesture. Place the GestureManager component on a form at design time and double click on it. This opens the Custom Gestures editor, featuring a list of custom gestures for the component. Of course, at the start of the process it will be empty.

Here you can see the Custom Gestures editor after I added a custom action (shaped like an 8), which I've defined in the CustomGestureTest example:



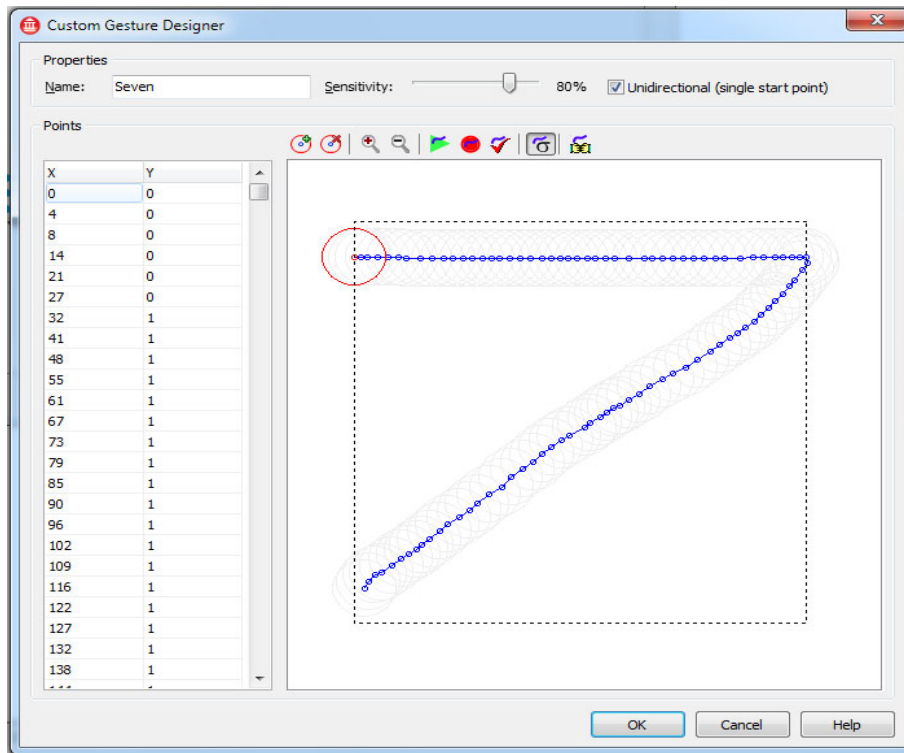
Now you can press the Create button to define a brand new gesture. This will open up the Custom Gesture Designer pane, in which you can graphically design your gesture (by dragging the mouse over the design surface) and also change some more global settings like the sensitivity or remove and insert specific points.

On the side of the gesture design surface, in fact, you can see the list of points that make up the gesture. This is very important, as this is the actual representation of the gesture itself: an array of points.

What is important, though, is that these points are not considered as absolute values, but what matters is their relative position. In practice, this means you can match the gesture both with an exaggerated or tight movement, but this movement has to be proportional to the original sequence of points.

Of course, a perfect match will be almost impossible, and this is where the Sensitivity factor comes into play. It determines the distance from each of the original points within which the gesture replay points must fall. This is both a variation in the distance or distortion from the original sequence, and also a distance of following the points in terms of speed. If you move too fast, too few points will be recorded and they won't fit in the area of the original ones.

I've recorded an image of the designer with a figure 7 shaped gesture below:



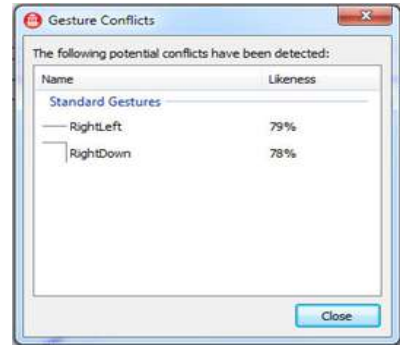
Again, this is not easy to describe in words and you can probably figure out a lot of details by experimenting directly with the Custom Gesture Designer⁶⁸.

Notice that this 7-shaped gesture conflicts with two others, as you can figure out in detail by clicking on the last icon above the designer. This opens up a list

⁶⁸ In particular, the gesture engine will use the sensitivity to rotate the gesture in case the gesture was drawn at a bit of an angle and it compares the angles between the gesture and the one drawn. Note also the unidirectional check box which let's you perform the gesture also reversed.

of conflicting gestures, showing the likeness of a conflict for each, as you can see on the side.

Now that we have added a couple of custom gestures to the GestureManager, we can figure out how they are saved in the program. If you look into the DFM file for the form, or copy the GestureManager component from the form and paste it in an editor, you'll see the actual data for the gesture. This includes the list of custom actions with their parameters and their list of points, plus the usual list of gestures enabled for each connected visual control, which includes both standard and custom gestures (the -1 gesture tied to Action1, in this case):



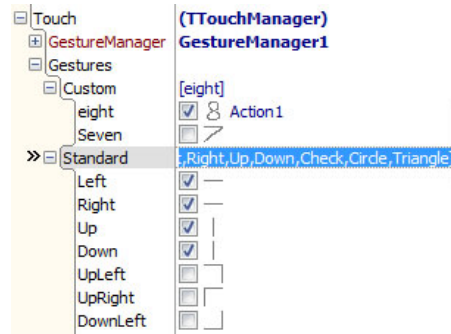
```
object GestureManager1: TGestureManager
  CustomGestures = <
    item
      Deviation = 34
      ErrorMargin = 46
      GestureID = -1
      Name = 'eight'
      Points = {... actual binary data ...}
    end
    item
      Deviation = 20
      ErrorMargin = 20
      GestureID = -2
      Name = 'Seven'
    end
  end>
  GestureData = <
    item
      Control = Panel1
      Collection = <
        item
          GestureID = sgi Left
        end
        ...
        item
          Action = Action1
          GestureID = -1
        end
      end
    end
  end>
end
```

Rather than be saved internally in the DFM, custom gestures can also be saved to an external file and referenced from the GestureManager, which can help you share the same gesture from multiple applications and edit them without having to recompile the program.

Having looked at the definition of the custom gestures and their storage, we can continue writing the code for this test application. The main form of the CustomGestureTest program has a large panel used to test the gestures, the GestureManager, plus a GestureListView and a GesturePreview.

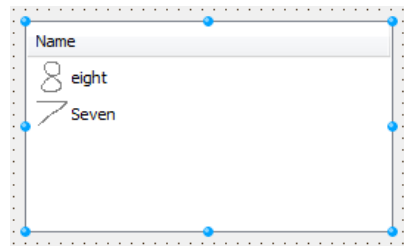
The panel handles both custom and standard gestures, as you can see by exploring the sub-properties of its Touch property, which includes the two separate lists, as you can see here on the side.

The panel manages a few gestures in its OnGesture event handler, while a couple of them are hooked to simple corresponding actions (PasteAction and Action1):



```
procedure TCustomGestureForm.Panel1Gesture(Sender: TObject;
const EventInfo: TGestureEventInfo; var Handled: Boolean);
begin
  case EventInfo.GestureID of
    sgi Left: Panel1.Caption := 'Left';
    sgi Right: Panel1.Caption := 'Right';
    sgi Up: Panel1.Caption := 'Up';
    sgi Down: Panel1.Caption := 'Down';
    sgi Check: Close;
  else
    Panel1.Caption := IntToStr (EventInfo.GestureID);
  end;
end;
```

Another component of the demo is the list of gestures, with a very limited preview. This is offered by the ready-to-use GestureListView control. As you drop it on a form at design time and connect it to the GestureManager, you'll automatically see its list of custom gestures, as you can see here for the example. As you run the application, you'll also see the same (limited) list at run time.



There are two ways to add standard gestures to the GestureListView control. One is to add individual gestures to the list, by calling for example:

```
GestureListView1.AddGesture(sgi Left);
```

Another is to add all of the gestures handled by a given target control, like in this case the panel:

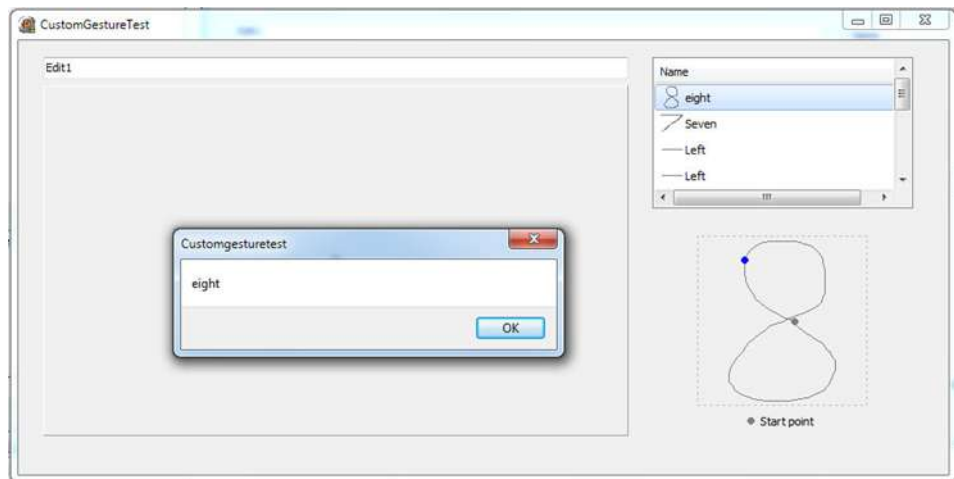
```
GestureListView1.AddGestures(Panel1);
```

Of course, if you call both (as I do in the demo program) and don't clear the list, you'll get duplicated entries. I've left those in the demo on purpose, but in the real world you'll probably want to call the `ClearGestureList` method before adding the list of gestures of a given control.

The last component of the demo program is a `GesturePreview`, which lets the user preview entries of the connected `GestureListView` control, hooked to the `GestureProvider` property:

```
object GesturePreview1: TGesturePreview
  Width = 250
  GestureProvider = GestureListView1
end
```

The output of the final program looks like the following⁶⁹:



Database Gestures

In trying to build a practical example of the use of gestures, I've decided to create a simple database application based on a `DBGrid`. My goal is to let users perform simple gestures (like up, down, left, and right) to move around the data set (performing the first, last, previous, and next actions, respectively).

⁶⁹ Again, it is very difficult to show the behavior of such an interactive application in an image: a video for this project is available on the book web site.

First, I created a simple application with a DBGrid, a DataSource, and a ClientDataSet connected one to the other:

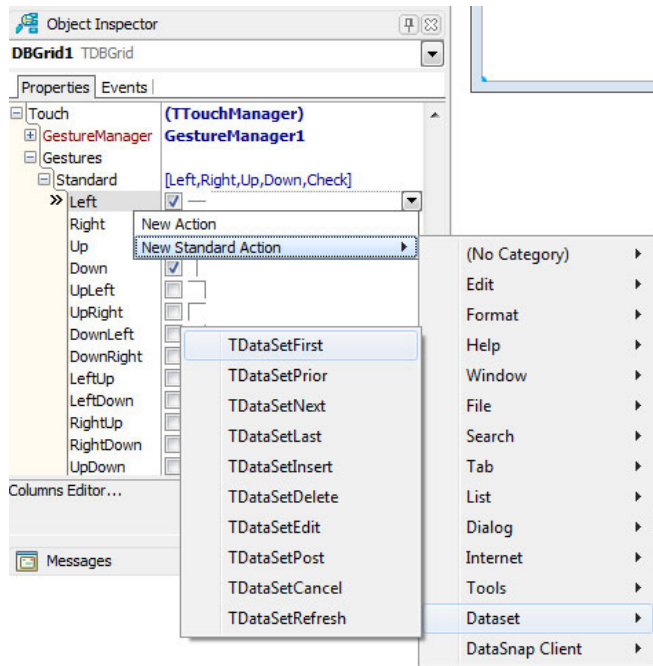
```
object ClientDataSet1: TClientDataSet
    FileName = '...\CodeGear Shared\Data\customer.cds'
end
object DataSource1: TDataSource
    DataSet = ClientDataSet1
end
object DBGrid1: TDBGrid
    DataSource = DataSource1
    Options = [dgTitles, dgIndicator, dgColLines,
        dgRowLines, dgRowSelect, dgConfirmDelete]
end
```

Next I've added a GestureManager component, an ActionList component, and an ImageList component to the form. Remember you have to connect the ImageList to the Images property of the ActionList if you want to add the pre-defined images for the standard actions to the program.

Now you can enable gestures for the DBGrid as usual, and customize the standard gestures by enabling them and creating the proper standard actions in place⁷⁰ in the Object

Inspector, as highlighted in this image.

What I have done to obtain this image was to expand the list of Standard gestures, enable one (Left) with the check box, click on the drop down combo box, pick the *New Standard Action* menu item, navigate to a given category (*Dataset*), and select the action I was interested in (TDataSetFirst).



⁷⁰ This technique was already mentioned in the section “Property Editors for Actions and Dates” of Chapter 5.

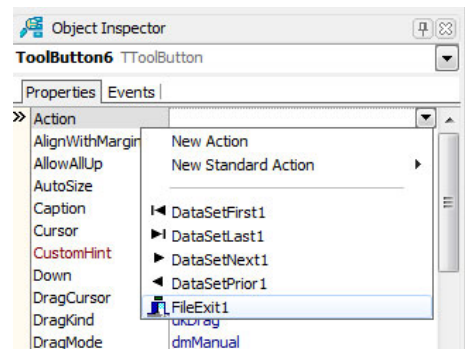
I ended up picking 5 actions, four movements plus one to close the application:

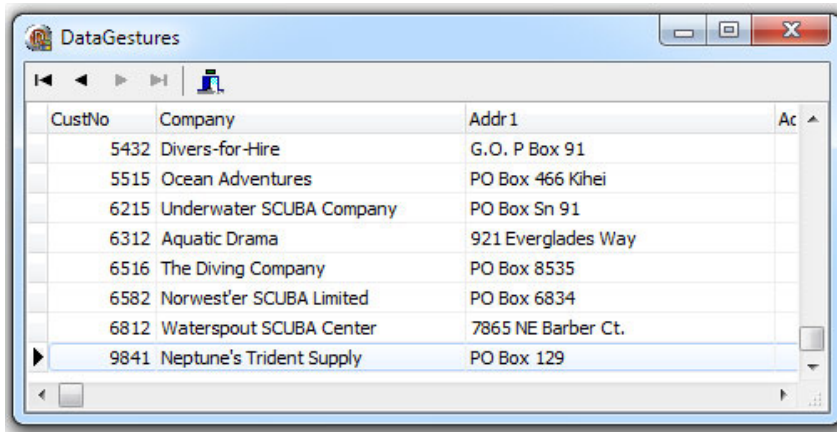
```
object GestureManager1: TGestureManager
  GestureData = <
    item
      Control = DBGrid1
      Collection = <
        item
          Action = DataSetPrior1
          GestureID = sgi Left
        end
        item
          Action = DataSetNext1
          GestureID = sgi Right
        end
        item
          Action = DataSetFirst1
          GestureID = sgi Up
        end
        item
          Action = DataSetLast1
          GestureID = sgi Down
        end
        item
          Action = FileExit1
          GestureID = sgi Check
        end
      end>
    end>
end
```

Next I added a Toolbar control to the application, created five buttons, hooked the existing actions to the toolbar buttons (using the Object Inspector as in the image here on the right).

To let users perform the actions on the toolbar area as well, I also added the same gestures of the DBGrid to the form. With the toolbar not having gesture support, the corresponding operations will be sent to the control or window behind it.

Finally, I added an Open call for the ClientDataSet control in the OnCreate event handler of the form, and I had a running application with basically no Delphi code behind it. Run the program, perform a “down” gesture, and the DBGrid will jump to the last record, as shown in the image of the next page.





This program has a significant problem, though. As you perform any gesture over the grid, it also receives regular mouse messages and selects the record under a mouse click. So if you do a “right” gesture, you’ll click and select the record under the mouse and also perform the gesture and move to the next record. This is not very intuitive. If you perform the gestures in the thin toolbar area, everything works as expected.

For a touch enabled application, though, we might want to disable the standard “click-to-select” grid operation, to force users to perform gestures (or use toolbar buttons) to move around the data. This can be achieved by disabling the mouse operations on the grid.

The simplest way to do this is to subclass the grid (using an interceptor class) and return False from the Focused virtual method, indirectly called at the beginning of each mouse operation. This is the code (available in the DataGestures example, but commented out in the source) you can use:

```
type
  TDBGri d = class (DBGri ds. TDBGri d)
  public
    function Focused: Boolean; override;
  end;

function TDBGri d.Focused: Boolean;
begin
  Result := False;
end;
```


Touch Keyboard

When you are creating applications for kiosks and other devices which don't have a keyboard attached, it is nice to show one on screen and let users type by selecting these keys with their input device (possibly their finger, as using a virtual keyboard with the mouse is far from a nice experience).

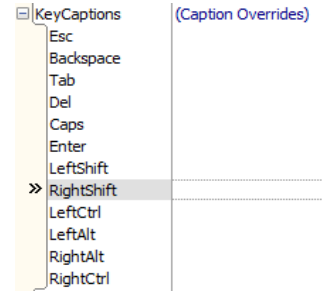
The VCL in Delphi 2010 includes a framework for creating virtual keyboards, based on the new TouchKeyboard component and the two related units, Keyboard and KeyboardTypes. If you drop the TouchKeyboard component on a form at design time you'll see something like in the following image:



Of course, what you'll see depends on your active keyboard at the operating system level. What I've shown here is the output when I set the keyboard to “en-US”. If I keep my standard settings, I'll generally see an Italian keyboard:



Notice that while punctuation characters are indeed replaced by the corresponding elements in Italian, special purpose keys (like Esc, Caps, Del) are not modified. The issue is that the captions of these keys are not defined at the operating system, so the only option is to override the defaults at the TouchKeyboard component level, using its `KeyCaptions` property you can use to change individual captions, as shown here.



Another key property of the component is its ability to display a numeric keypad (like the one on the right) rather than a full keyboard, by changing the value of the `Layout` property (a string, not an enumeration, to allow future expansion and custom keyboard layouts) from *Standard* to *Keypad*⁷¹.



Regardless of the layout, the effect of using this virtual keyboard is to send input to the control currently having the input focus, like a physical keyboard. Of course, this means that the button representing keys won't get the focus when pressed, preserving the current input focus. You can see this effect in the basic KeyboardTest application (here when holding the Shift key):



71 You can create custom layouts, as Chris Bensen started explaining on his blog in the post <http://chrisbensen.blogspot.com/2009/12/hacking-ttouchkeyboard-part-i.html> and in the following 3 parts, showing also how to create a custom layout from an XML representation of the keyboard.

The program also has a button that I've used to make a few experiments and try to find a workaround to an annoying VCL problem. As soon as I run a standard VCL application (including Delphi itself) it will reset my current keyboard to the default one. So whatever my keyboard setting a Delphi application will start with the Italian keyboard⁷².

In the button I've made two experiments, as you can see I the code below:

```
procedure TKeyboardForm.btnTestClick(Sender: TObject);
var
  code: UI NT;
begin
  LoadKeyboardLayout ('00000409', KLF_ACTI VATE);

  code := MapVi rtual Key(43, MAPVK_VSC_TO_VK);
  ShowMessage (IntToStr (code));
end;
```

First, I write the code need to force a different keyboard, specifically the '*en-US*' keyboard, which has the keyboard layout code '*00000409*'. Loading and activating this layout broadcasts a `wm_InputLanguageChange`, which the `TouchKeyboard` component handles updating its layout. As an alternative you can call the `ActivateKeyboardLayout` function and `Redraw` the touch keyboard after letting the application process update messages:

```
ActivateKeyboardLayout(67699721 {en-US}, 0);
Application.ProcessMessages;
TouchKeyboard1.Redraw;
```

Second, I use the same code of the `TouchKeyboard` component for converting a virtual key into an actual character code, as an experiment. I used this code to figure out the VCL issue described earlier.

Overall the `TouchKeyboard` component can be handy in a kiosk or similar application, while in other occasions you might want to hide it and display it on request. Of course, you'll always need to have it on screen only while the user has to input some data, or you can prepare a standard "keyboard entry form" with a single edit box and a keyboard you show every time there is an input request (like when the actual edit in the main form received the focus). I've not created a demo program showing a similar situation, but it shouldn't be terribly difficult to create one.

72 The situation is actually worse, as running a Delphi program changes the active keyboard at the operating system setting, that is for each running application. Quite annoying, although few people use multiple settings.

Multi-Touch Support

As I detailed in the section “From Single Touch to Multi-Touch”, Windows 7 and Delphi 2010 have specific support for multi-touch hardware, basically in the form of a new Windows message (`wm_touch`) and support for handling it at the VCL level. Considering the current limited crop of multi-touch enabled hardware, though, this support should probably be restricted to specific applications for quite some time. This is why the coverage of native touch, or `wm_touch`, in this chapter is restricted to this section and based on an example I've borrowed from Chris Bensen⁷³.

Before we get specifically to `wm_touch`, consider that some information about touch-based requests is also surfaced in traditional mouse events. Pressing on the screen (or touchpad) with your finger results in a VCL *mouse down* event, carrying over information about the source. This is provided in the `Shi ftState` parameter of mouse events, as the `TShi ftState` enumeration has been extended with two new elements, `touch` and `pen`. The enumeration in Delphi 2010 has the following values:

```
TShi ftState = set of (ssShi ft, ssAl t, ssCtrl,
    ssLeft, ssRi ght, ssMi ddl e, ssDoubl e, ssTouch, ssPen);
```

This is relevant because you might want to handle touch operations like mouse operations, providing limited extra information. On the other hand, if you are specifically handling touch operations, you might want to disable any mouse request coming from a touch source, or you'll handle the request twice (the first time as a native touch request, and the second time as a mouse operation originating from the touch request).

Handling `wm_touch`

The `wm_touch` message is a raw, low-level Windows message, providing a significant amount of information about the user input (unlike a mouse message). Considering the limited data a Windows message can carry, this is actually not a precise description.

⁷³ Chris is a Delphi R&D member who worked on touch support in Delphi 2010. His blog is at <http://chrisbensen.blogspot.com/>. I got permission from him to quote the source code of this demo application in my book.

What you receive in the message's `LParam` is the handle of a touch information data structure you can retrieve by calling the `GetTouchInputInfo` API function. Before you call this function, you must allocate the proper amount of memory for the array of touch points, using the number of points that is carried in the `WParam` of the message. At the end, you must release the touch information data structure by calling the `CloseTouchInputHandle` API function.

Here is how the standard `wm_Touch` handling code looks like:

```
// in the TMyForm class declaration
procedure WMTouch(var Message: TMessage); message wm_Touch;

procedure TMyForm.WMTouch(var Message: TMessage);
var
    TouchInputs: array of TTouchInput;
    TouchInput: TTouchInput;
begin
    SetLength(TouchInputs, Message.WParam);
    GetTouchInputInfo(Message.LParam, Message.WParam,
        @TouchInputs[0], SizeOf(TTouchInput));
    for TouchInput in TouchInputs do
        ...
    CloseTouchInputHandle(Message.LParam);
end;
```

Each `TouchInput` element of the array is a record with the information about the touch position (x and y) in “hundredths of a pixel of physical screen coordinates”⁷⁴, a handle to the input device, an ID of the touch point (which remains the same over time while the user keeps pressing or moving a finger), several flags indicating the current operation for the touch point (up, down, move, and so on), the time stamp of the operation, the horizontal and vertical size of the touch or contact area:

```
type
    TOUCHINPUT = record
        x: Integer;
        y: Integer;
        hSource: THandle;
        dwID: DWORD;
        dwFlags: DWORD;
        dwMask: DWORD;
        dwTime: DWORD;
        dwExtraInfo: ULONG_PTR;
        cxContact: DWORD;
        cyContact: DWORD;
    end;
```

74 According to Microsoft's SDK documentation for the data structure at: <http://msdn.microsoft.com/en-us/library/dd317334.aspx>

To manage and interpret the x and y coordinates, you might want to call the `PhysicalToLogicalPoint` Windows API function to convert the physical screen location to the logical coordinates the application understands.

To receive this information, as mentioned, you need to have multi-touch enabled hardware, but the application must also register individual windows to receive touch messages by calling the `RegisterTouchWindow` API function. You'll make this call after the form handle has been created, for example in an overridden version of the `CreateWnd` method:

```
procedure TMyForm.FormCreate (Sender: TObject);
begin
  inherited;
  RegisterTouchWindow(Handle, 0);
end;
```

Remember to also *unregister* the window for touch, once you've finished with it, by calling the symmetric API function `UnregisterTouchWindow`⁷⁵.

Chris Bensen's TouchMove Demo

To demonstrate touch support in Delphi 2010, along with `Direct2D` support (that I covered in the last chapter) and inertia manipulations (that I'll cover in the next section), Chris Bensen wrote a very nice demo called `TouchMove`, which I decided to refer to in this section. I won't cover other elements of the demo, only how it manages touch⁷⁶. The demo handles a number of touch points at the same time, creating for each a “glow spot” element or using an existing one if the touch operation was performed on active spot.

For doing any point-based manipulation, the code uses a support function (`TouchPointToPoint`) to convert the coordinates of the touch points from hundredths of a pixel to pixels and from device-based points to logical points:

```
function TouchPointToPoint(const TouchPoint: TTouchInput): TPoint;
begin
  Result := Point(TouchPoint.X div 100, TouchPoint.Y div 100);
  PhysicalToLogicalPoint(Handle, Result);
end;
```

⁷⁵ Unregistering touch windows on termination is nice, but doesn't seem to be required. According to MSDN documentation, you should call `UnregisterTouchWindow` to indicate that a window “no longer requires touch input”. That's all.

⁷⁶ For the links to his four-part description of the `TouchMove` demo see Chris' summary post see <http://chrisbensen.blogspot.com/2009/11/touch-demo.html>

The core of the program is in his `wm_touch` message handler.

```

procedure TTouchForm.WMTouch(var Message: TMessage);
var
    TouchInputs: array of TTouchInput;
    TouchInput: TTouchInput;
    Handled: Boolean;
    Point: TPoint;
    TouchMessage: TTouchMessage;
begin
    Handled := False;
    SetLength(TouchInputs, Message.WParam);
    GetTouchInputInfo(Message.LParam, Message.WParam,
        @TouchInputs[0], SizeOf(TTouchInput));
    try
        for TouchInput in TouchInputs do
            begin
                Point := TouchPointToPoint(TouchInput);
                if (TouchInput.dwFlags and TOUCHEVENTF_MOVE) <> 0 then
                    TouchMessage := tmMove
                else if (TouchInput.dwFlags and TOUCHEVENTF_UP) <> 0 then
                    TouchMessage := tmUp
                else if (TouchInput.dwFlags and TOUCHEVENTF_DOWN) <> 0 then
                    TouchMessage := tmDown;
                ProcessTouchMessages(Point, TouchInput.dwlD, TouchMessage);
            end;
            Handled := True;
        finally
            if Handled then
                CloseTouchInputHandle(Message.LParam)
            else
                inherited;
        end;
    end;

```

As you can see in the `WMTouch` method, the code grabs information about the touch operations picking a value out of the `TTouchMessage` enumeration, which is later passed to the `ProcessTouchMessages` method. The goal is to use the same processing function, as for mouse operations (which are processed only if they didn't originate from a touch action, to avoid any double processing), as for example in the `OnMouseDown` handler:

```

procedure TTouchForm.FormMouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    if ssTouch in Shift then
        Exit;
    FMouseDown := True;
    ProcessTouchMessages(Point(X, Y), 0, tmDown);
end;

```

In this case there is no need to convert the coordinates, as they are exactly what the program expects. Finally, the `ProcessTouchMessages` method looks for

an existing *glow spot* with the given ID or one at the given location, and if none is found creates a new one:

```
function TTouchForm.ProcessTouchMessages(const APoint: TPoint;
  ID: Integer; TouchMessage: TTouchMessage): T GlowSpot;
var
  Spot: T GlowSpot;
begin
  Result := nil;

  Spot := FindSpot(ID); // find by ID
  if Spot = nil then
  begin
    Spot := FindSpot(APoint); // find by location
    if Spot <> nil then
      Spot.ID := ID;
  end;

  if Spot = nil then // create a new one
  begin
    Spot := T GlowSpot.Create(Self);
    Spot.ID := ID;
    FSpots.Add(Spot);
  end;

  Spot.DoTouch(APoint, ID, TouchMessage);
  Result := Spot;
end;
```

Again, there is much more to this demo, of which I didn't display any images as they are very hard to capture on paper. The complete source code of Touch-Move example is included in the book source code and it is also available at the original download location:

■ <http://cc.embarcadero.com/item/27469>

Inertia Manipulation (with no Touch)

The demo by Chris Bensen makes a very interesting use of the Manipulations and Inertia engine that's built into Windows 7 and surfaced in corresponding Delphi API interface units. This engine is based on COM, and is ready to use from Delphi applications. All you have to do is to figure out how, and most of the available demos are quite complex. So I've decided to write a program specifically focused on using the Inertia support, with no touch, no Direct2D, and

no other features that would distract you. All this program does is show a ball (a colored circle) and lets you move, throw it... and see it bounce.

At the core of the InertiaBall example there is a `TBall` class representing a single object (I haven't extended the application to handle multiple objects at once to keep it as simple as possible). Internally this class uses the inertia processor and the manipulation processor provided by Windows 7, and implements a specific manipulations interface, used as a callback to notify changes in the position of the object.

- The **manipulation processor** lets you interact with a physical object represented on the screen in a more realistic way and gives access to its status. For example, by asking the manipulation processor to change the position of an object we'll be able to ask for the current speed of the object.
- The **inertia processor** lets you implement a realistic behavior for an object, like keep it moving at its current speed, slowed down by its natural inertia, or let it bounce against the borders of a surface.

For more information on these two COM objects available in Windows 7 and their interfaces (available in Delphi 2010 in the Manipulations unit), you can refer to the API documentation on the MSDN site at:

■ <http://msdn.microsoft.com/en-us/library/dd372613.aspx>

After this short introduction, this is the declaration of the class representing a bouncing ball:

```
uses
  Manipulations;
type
  TBall = class (TInterfacedObject, _IManipulationEvents)
  private
    FInertia: IInertiaProcessor;
    FManipulator: IManipulationProcessor;
```

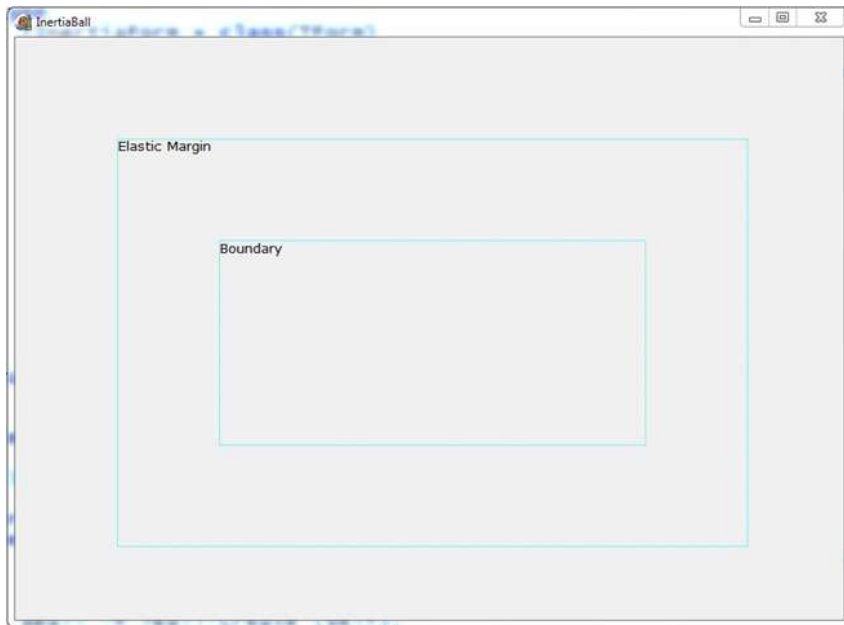
As the program creates an object of this class (which happens as the main form is itself created), it creates the two COM objects, sets the interface of the object as a response to the COM object events (handling the `_IManipulationEvents` interface), and initializes a few properties of the inertia manipulator:

```
constructor TBall.Create (aForm: TForm);
begin
  inherited Create;
  ID := 1;
  FCompleted := True;
  FInertia := CreateComObject(CLSID_IInertiaProcessor)
    as IInertiaProcessor;
  FManipulator := CreateComObject(CLSID_IManipulationProcessor)
    as IManipulationProcessor;
```


192 - Chapter 6: Touch and Gestures

```
InterfaceConnect(FInertia, _IManipulationEvents,  
    Sel f, FInertiaCookie);  
InterfaceConnect(FManipulator, _IManipulationEvents,  
    Sel f, FManipulatorCookie);  
  
FInertia.put_DesiredDeceleration(0.001);  
FInertia.put_BoundaryLeft(200);  
FInertia.put_BoundaryTop(200);  
FInertia.put_BoundaryRight(aForm.Width - 200);  
FInertia.put_BoundaryBottom(aForm.Height - 200);  
FInertia.put_ElasticMarginLeft(100);  
FInertia.put_ElasticMarginTop(100);  
FInertia.put_ElasticMarginRight(100);  
FInertia.put_ElasticMarginBottom(100);  
end;
```

As you can see the movements boundary (called *Boundary* in the image) is set at 200 pixels within the border of the form. However, half of this area (100 pixels all around) is actually used as an elastic, bouncing area (called *Elastic Margin* in the image). Outside of this margin (which is external to the boundary) there is an area the object will never reach⁷⁷, which is 100 pixels wide. The three areas are depicted in the image of the next page, which is an actual image of the program from which I've removed the bouncing ball.



⁷⁷ The object will never reach the area after you *throw* it, but the program doesn't prevent you from dragging it to this external area...

As I mentioned, the `TBall` object is created in the middle of the form as the program starts (after tweaking the division by zero system exception⁷⁸):

```
procedure TIInertiaForm.FormCreate(Sender: TObject);
begin
    // disable div by 0 exceptions for the inertia processor
    Set8087CW($133F);

    aBall := TBall.Create (self);
    aBall.X := Width div 2;
    aBall.Y := Height div 2;
    aBall.Radius := 20;
    aBall.Color := clRed;
    aBall.ID := 1;
end;
```

The ball is painted on the screen along with the two focus rectangles in the `OnPaint` event handler of the form:

```
procedure TIInertiaForm.FormPaint(Sender: TObject);
begin
    aBall.Paint(Canvas);
    DrawFocusRect(Canvas.Handle,
        Rect(100, 100, Width-100, Height-100));
    DrawFocusRect(Canvas.Handle,
        Rect(200, 200, Width-200, Height-200));
end;
```

The actual painting of the ball is quite trivial:

```
procedure TBall.Paint(Canvas: TCanvas);
begin
    Canvas.Brush.Color := self.Color;
    Canvas.Ellipse(x-Radius, y-radius, x+radius, y+radius);
end;
```

Things start getting interesting as you move the mouse. In this case rather than changing the ball position directly, the program changes it via the manipulation processor, which will notify the ball using the callback events of the `_IManipulationEvents` interface. The three manipulation operations take place as the user presses the mouse button, moves the mouse, and releases it. Each of these three events at the form level calls a corresponding event of the ball object, which calls a corresponding method of the manipulation processor. It is worth following each of them, to understand the temporal sequence (as it took me a while to figure it out).

78 If you don't disable the *div by zero exception*, the inertia processor will indeed throw them quite often, as you can easily figure out by commenting out that line of code and running the program. How did I find out? By looking at the Inertia processing demos from MSDN, after hitting way too many errors with my code.

The first operation is a mouse down at the form, ball, and manipulation manager level:

```
procedure TInertiaForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  aBall.MouseDown(X, Y);
  Invalidate;
  FMouseDown := True;
end;

procedure TBall.MouseDown(X, Y: Integer);
begin
  FManipulator.ProcessDown(ID, X, Y);
end;
```

The program will receive several mouse move events:

```
procedure TInertiaForm.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
begin
  if FMouseDown then
    begin
      aBall.MouseMove(X, Y);
      Invalidate;
    end;
end;

procedure TBall.MouseMove(X, Y: Integer);
begin
  FManipulator.ProcessMove(ID, X, Y);
end;
```

The last (and most important) operation is the mouse up. At the end of this manipulation, in fact, the program starts the inertia processor:

```
procedure TInertiaForm.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  aBall.MouseUp(X, Y);
  FMouseDown := False;
  Invalidate;
  Timer1.Enabled := True;
end;

procedure TBall.MouseUp(X, Y: Integer);
var
  vx, vy: Single;
begin
  FManipulator.ProcessUp(ID, X, Y);

  FManipulator.GetVelocityX(Vx);
  FManipulator.GetVelocityY(Vy);
  FIInertia.put_InitalVelocityX(Vx);
  FIInertia.put_InitalVelocityY(Vy);
```

```

    FI nerti a. put_Ini ti al Ori gi nX(X);
    FI nerti a. put_Ini ti al Ori gi nY(Y);

    FCompl eted := Fal se;
end;

```

The inertia processor doesn't do much on its own. You have to ask it often to process its status (which is clearly time dependent), for example using a timer:

```

procedure TI nerti aForm. Ti mer1Ti mer(Sender: TObj ect);
begin
    aBal l. ProcessI nerti a;
    I nval i date;
end;

procedure TBal l. ProcessI nerti a;
begin
    if not FCompl eted then
        FI nerti a. Process(FCompl eted);
    end;

```

Each time the manipulation or inertia processors compute a new position for the object, they notify it via the `Mani pul ati onDel ta` method of the `_I Mani pul ati onEvents` interface:

```

functi on TBal l. Mani pul ati onDel ta(X, Y,
    transl ati onDel taX, transl ati onDel taY,
    scal eDel ta, expansi onDel ta, rotati onDel ta,
    cumul ati veTransl ati onX, cumul ati veTransl ati onY,
    cumul ati veScale, cumul ati veExpansi on,
    cumul ati veRotati on: Si ngl e): HRESULT;
begin
    sel f.X := Round (X);
    sel f.Y := Round (Y);
    Resul t := S_OK;
end;

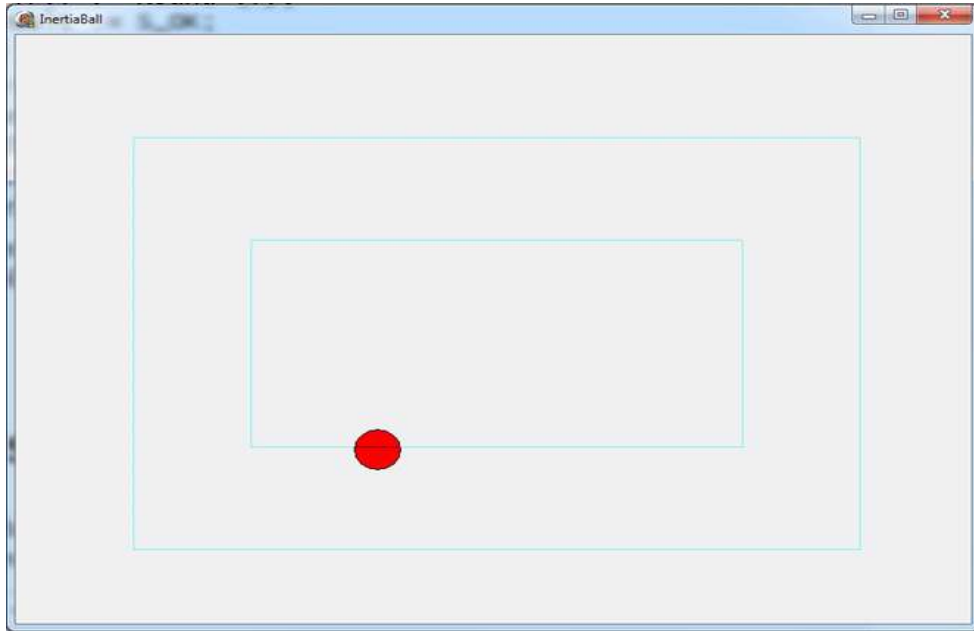
```

Notice that all manipulations and inertia information uses `Si ngl e` coordinates for much better precision⁷⁹, which is why these examples are better built using the `Direct2D` output and its floating point coordinates, but again I wanted to keep this as simple as possible, so I used a traditional GDI painting surface.

The result cannot be easily depicted in a static image, although you can see one in the next page, and can be appreciated much better by running the program⁸⁰.

79 In a first tentative attempt, I added to the current X position the `transl ati onDel taX`, but rounding this value it soon got down to zero, practically stopping the operation well before it was done.

80 Or by looking at the project video, again available on the book web site.



What's Next

This chapter ends the section of the book covering the VCL and Windows 7. We have discussed many new Windows 7 APIs, some of the corresponding new VCL controls, and other new features of the VCL which are only partially related to Windows 7 and will also work perfectly on older versions of Windows, like gestures.

In the last two chapters of the book, I'll focus on Database support, mostly in multi-tier and Web applications based on the REST protocol. The main focus of the two chapters, in fact, is to cover new extensions of the DataSnap engine, although there is also some coverage of the dbExpress framework and of other data access technologies that Delphi 2010 provides.

Chapter 7:

Database Access

And DataSnap

The development of database-oriented applications, with stand-alone, client/server, or multi-tier architectures, has always been one Delphi's strong features. It is not surprising that this version brings incremental new features to the database area of the product.

While Delphi 2007 introduced a new generation of dbExpress (called dbExpress IV) and Delphi 2009 brought us a new generation of DataSnap (often referred to as DataSnap 2009), Delphi 2010 provides incremental features in both areas. Specifically regarding the DataSnap multi-tier model, this version completes the new architecture that we could say was only partially available in Delphi 2009. New features include HTTP support, callbacks, filtering with compression and encryption, and REST interfaces.

Notice that if you've never used these two technologies (dbExpress and DataSnap) you won't find introductory material here, but mostly a focus on what's

new in Delphi 2010. Consider also that DataSnap and the complete versions of dbExpress, with all new drivers and connectivity to remote databases, are available only in the Enterprise and Architect versions of Delphi.

However, before I cover these two main features, let me touch on a few other interesting new elements of Delphi's database architecture.

New Field Types and Other Core Database Extensions

Although the enhancements to the overall Delphi database architecture can be considered as minor, the new features that have been introduced will have a very significant impact for some developers. For example, for managing floating point numbers with a limited representation and time stamp offsets, there are specific `TFIELD`-derived classes that you can now use:

```
TSingleField = class(TNumercialField)
TSQLTimeStampOffsetField = class(TSQLTimeStampField)
```

Support for time stamp offset processing is now also available in the new `TSQLTimeStampOffset` class that's been added to the `SqlTimSt` unit.

Matching these two new field types, plus the `TObjectField` types now used by DataSnap, there are three new elements in the `TFIELDType` enumeration: `ftTimeStampOffset`, `ftObject`, and `ftSingle`.

Also, in the `TFIELD` class you can convert the value of the current element using some new *As* properties:

```
property AsSQLTimeStampOffset: TSQLTimeStampOffset ...
property AsSingle: Single ...
property AsLargeInt: LargeInt ...
```

Corresponding *As* properties have been added to the `TParam` class. Given the new field classes introduced in Delphi 2010, the hierarchy of the `TFIELD` classes defined in the `DB` unit becomes even bigger.

To help you get a full picture, I've provided a complete class tree in the following page (with new classes introduced in Delphi 2010 marked in bold).

```

TFi el d
  TStringFi el d
    TWi deStringFi el d
    TGui dFi el d
  TNumeri cFi el d
    TI ntegerFi el d
      TAutoI ncFi el d
      TSmal l i ntFi el d
      TShorti ntFi el d
      TByteFi el d
      TWordFi el d
    TLongWordFi el d
      TUnsi gnedAutoI ncFi el d
    TLargei ntFi el d // Int64
    TFI oatFi el d
      TCurrencyFi el d
    TExtendedFi el d
    TBCDFi el d
    TFMTBCDFi el d
    TSi ngl eFi el d
  TBool eanFi el d
  TDateTi meFi el d
    TDateFi el d
    TTi meFi el d
  TSQLTi meStampFi el d
    TSQLTi meStampOffsetFi el d
  TBi naryFi el d
    TBytesFi el d
    TVarBytesFi el d
  TBl obFi el d
    TMemoFi el d
    TWi deMemoFi el d // wi destring memo
    TGraphi cFi el d
  TObje ctFi el d
    TADTFi el d // Abstract Data Type
    TArrayFi el d
    TDataSetFi el d
    TReferenceFi el d
  TVari antFi el d
  TI nterfaceFi el d
    TI Di spatchFi el d
  TAggregateFi el d

```

There are some other changes focused on very specific needs, like local dependent string formatting for date and time functions, the remapping of 64-bit integers to the Int64 rather than BCD, and other similar changes probably not worth covering in detail.

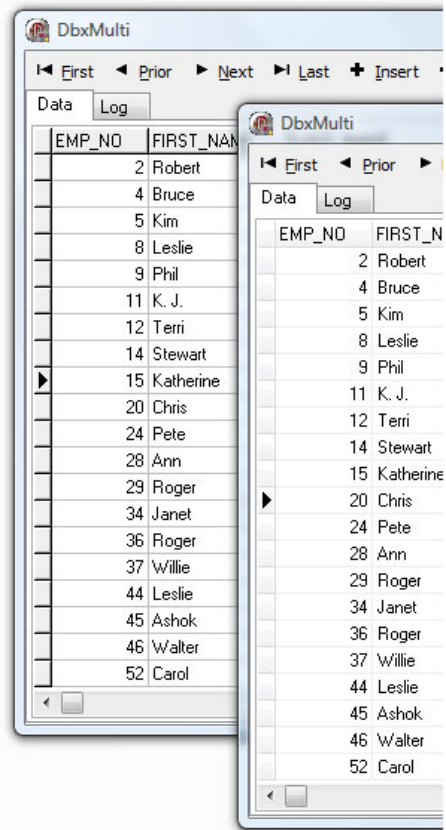
Themes Support and Other DBGrid Extensions

Even with the stronger themes support already in the VCL for a couple of versions (introduced in Delphi 7, but much expanded in Delphi 2007), not all of the visual controls got full themes support. A notable absence was in the VCL grids, both the plain ones (StringGrid and DrawGrid) and the data-aware version (DBGrid). This has now been addressed.

In the image on the side you can see the same application (DbxMulti2010, the new version of a simple dbExpress demo I used in previous books) with and without themes enabled. The difference is quite striking, as with minimal user interface changes the output looks more modern.

In case you don't want to use this new user interface style and still have the application themed, you can use the `DrawnStyle` property, which is set to `gdsThemed` by default but can be switched back to `gdsClassic` or the alternative `gdsGradient`. Along with the new gradient style, the grid controls have a `GradientStartColor` and a `GradientEndColor` property.

As an aside, the DBGrid control has two new options, `dgTitleClick` and `dgTitleHotTrack`. The `dgTitleClick` option lets you control whether the user can click on the title (eventually disabling the corresponding graphical effect). When this option is disabled, the `OnTitleClick` event won't fire, of course. Now the problem is that if you create a brand new application the `dgTitleClick` option is enabled by default. While for an existing application opened in Delphi 2010, it will be set to `False` as it was not in among the flags in the original DFM file. I guess they could have entered an option with the



opposite meaning (disable title click) to preserve compatibility, but it would have looked quite awkward⁸¹.

DBGrid In-place Editor Issues

The problem with any version of the `DrawnStyle` property is that as the DBGrid displays the in-place editor, it draws a thick black border around it. In effect, that's not a black border but the window background color, which is managed in a different way from the past. You can see the effect in the image here on the right.

How can we fix the problem⁸²? The issue is rooted deep in the internal painting code, which is extremely complex. There wasn't a single specific fix that caused this unwanted effect, but a set of changes related with themes support. The problem occurs for any themed application, as the issue shows up regardless of the value of the `DrawnStyle` property. We cannot even handle the `OnDrawDataCell` event of the DBGrid, as for cells under the in-place editor it doesn't get called. So we have to resort to a changing the internal behavior of the DBGrid and its `TCustomGrid` ancestor class.

EMP_NO	FIRST_NAME	LAST_NAME
2	Robert	Nelson
4	Bruce	Young
5	Kim	Lambert
8	Leslie	Johnson
9	Phil	Forest

I noticed that if we let the `Paint` method of the `TCustomGrid` class paint the specific cell in the same way it does for any other cell, the problem disappears. For a test, you can copy the Grids unit to a project; inside that `Paint` method you'll find the internal `DrawCells` subroutine (starting at line 2150); locate the test which determines if standard painting code is executed (at line 2186); you'll see the following:

```
if not (gdFocused in DrawState) or not (goEditing in Options) or
not FEditorMode or (csDesigning in ComponentState) then
```

If you comment out this code, the grid behavior will get back to normal. However, I don't particularly like changing VCL units, so I tried looking for an alternative fix. The idea is that if we make the test above succeed, standard

81 The problems with the `dgTitleClick` option were first reported by Bob Swart on his blog at <http://www.bobswart.nl/Weblog/Blog.aspx?RootId=5:3791>

82 I reported this bug (or significant change in behavior) on Quality Central and it is available at <http://qc.embarcadero.com/wc/qcmain.aspx?d=80209>

painting will take place also for the cell showing the in-place editor, and its background (visible only for the portion around the editor) will change from black to white or whatever is the correct color.

Armed with the idea and considering that code is called inside the `Paint` method, I decided to override it. As it is a virtual function, you cannot use a class helper, but you can use an *interposer* class⁸³. In this case, I've written a new unit with the following interface:

```
unit DbGridFix;

interface

uses
  DBGrids;

type
  TDBGri d = class (DbGrids. TDBGri d)
  protected
    procedure Paint; override;
  end;
```

As long as the unit is included after the `DBGrids` unit in any form that uses a `DBGri d` component, the compiler will refer to this version of the `TDBGri d` class rather than the official one. This means at design time you'll be using a standard `DBGri d`, while at run time the component is fully replaced by the version defined in the `DbGridFix` unit.

Inside the `Paint` method what we can do is to temporarily change the value of the `FEdi torMode` field, which determines the different behavior for the repainting of the area behind the in-place editor. This field is not used anywhere else in the painting code, so we should not cause any other problems. However, if we set the `Edi torMode` property, this will cause side effects (including a asking for a repaint of the cell, which will then triggers another repaint going on forever).

The solution would be to change the field, rather than property, but how can we accomplish this? In the past, considering it is a private field, we'd have to use some low-level hacks (accessing to a specific memory location), while now we

83 An interposer class is a class with the same name of the class it inherits from. By adding its unit after the unit of its base class in a `uses` statement, the program will use the modified version of the class rather than the original. Interposer classes are not a terribly neat technique, rather a hack. But they can be very handy, indeed! Historically, the name of the technique was given in the first article that described it, which appeared many years ago in "The Delphi Magazine".

can use the extended RTTI to access the private field). Still a low level technique, but neater and more flexible in case of future internal changes in the class structure.

Here is the complete code of the `Paint` method, which stores the current value of the field (directly using its `TValue`) and restores it at the end:

```
uses
  Rtti;

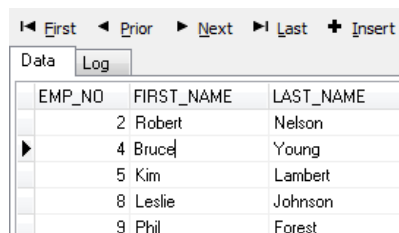
procedure TDBGrid.Paint;
var
  oldEditorMode: TValue;
  context: TRttiContext;
  editorModeField: TRttiField;
begin
  editorModeField := context.GetType(TDBGrid).
    GetField('FEditorMode');

  if Assigned(editorModeField) then
  begin
    oldEditorMode := editorModeField.GetValue(self);
    editorModeField.SetValue(self, TValue.From(False));

    // now paint
    inherited;

    if Assigned(editorModeField) then
      editorModeField.SetValue(self, oldEditorMode);
  end;
```

At the core the method calls the base class version of the method, as managing the actual painting of the grid would be daunting (that is, doing a copy and paste of hundreds of lines of source code). The effect of this fix is to get back the white area around the in-place editor, shown here, as was standard in past versions of Delphi:



EMP_NO	FIRST_NAME	LAST_NAME
2	Robert	Nelson
4	Bruce	Young
5	Kim	Lambert
8	Leslie	Johnson
9	Phil	Forest

Midas DLL Now With Source

A significant change “behind the scenes” in Delphi 2010 is the availability of the source code of `midas.dll`. This library, which is either deployed along with your application or compiled into the `Midaslib` unit and bound to your executable, is the engine behind the `ClientDataSet` component. In fact, even if there is the full Delphi source code of this component, its actual data processing code is inside the DLL, that the component invokes frequently.

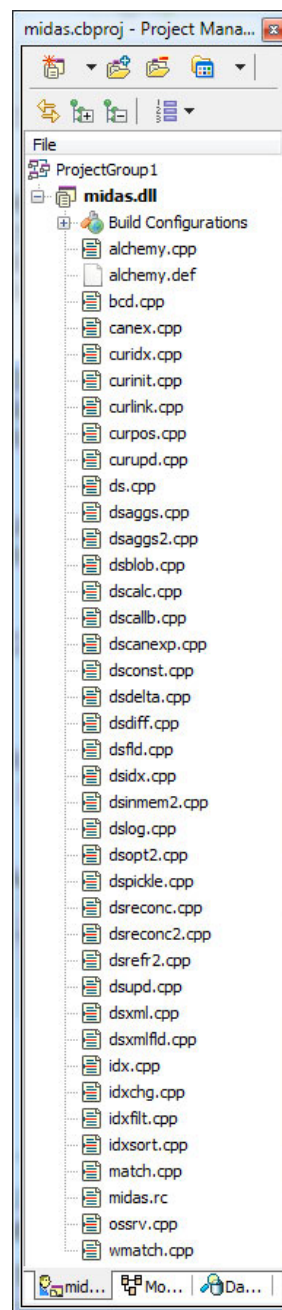
It happened over the last few years that developers spotted a couple of bugs in the DLL, which were very hard to find and fix given the source code was not available. Following complaints by the Delphi community, Embarcadero has finally decided to release the source code to the public, along with the VCL source code⁸⁴.

In my installation, the library source code is in:

```
C:\Program Files\Embarcadero\
RAD Studio\7.0\source\db\midas
```

The drawback, though, is that this library is not written in Delphi, but it is a C++ library. If you look at the code, you'll easily realize that it is far from easy to navigate and study it. Once opened in C++Builder, the project structure looks like the one of the entire side of this page.

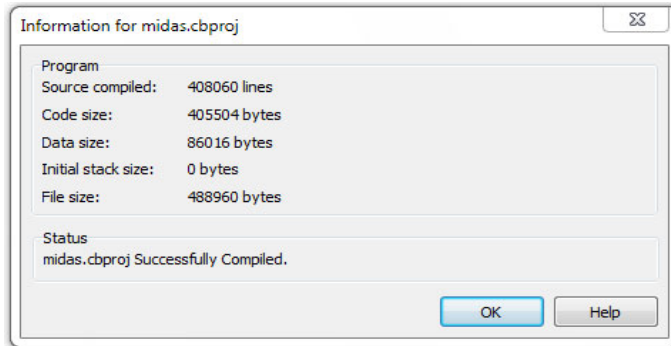
The main header files are `alchemy.h` and `alchemy.h`, which define the data set classes. The class structure is very simple, but the code is far from trivial and is almost 1.5 MB. Other header files and core implementations apparently come straight out of the Borland Database Engine, with references to the `IDAPIxx.DLL`, headers like `bdtypes.h`, and the like.



⁸⁴ Just one example of the improved community relations with Embarcadero.

BDE in-memory tables were probably a precursor of the ClientDataSet component, so this should be no surprise.

I didn't do much exploration of the Midas.dll source code, but I took a little time to configure the build environment and compile it:



As you can see, the library has over 400,000 lines of source code, and although the resulting DLL is slightly bigger than the one that ships with Delphi 2010, it seems to behave in the same way.

Overall, the fact that this source code is available is very interesting, even if very few Delphi developers would probably work on it directly. What is relevant is that some Delphi and C++ experts would have a chance to debug the library and make the component faster. For any Delphi users, having the source code of a core component provides a big guarantee in case a critical bug arises, as you don't have to wait for Embarcadero to fix it. In practice this already has happened with previous informal fixes (mostly done by Andreas Hausladen⁸⁵) now embedded in the current version of the library.

From a theoretical perspective, it would be much better to have a new ClientDataSet component rewritten at a higher level in Delphi and based on the internal memory manager, but given this would take quite some effort and it could easily cause incompatibilities with existing code, the current option to make available the Midas DLL source code written in C++ is a good solution.

⁸⁵ Again, Andreas blog is at <http://andy.jgknet.de>.

ADO 2.8 Support

Another change that's very specific to a given set of data access components is the upgrade to the latest version of the ADO library, to support version 2.8. As you can see in the ADOInt unit, the library versions are declared as:

```
ADODBMajorVersion = 2;  
ADODBMinorVersion = 8;
```

There are many other changes in the unit, with the declaration of newer internal interfaces and some new constants. There is no significant change in the way these interfaces are called by the ADODataset component, but you'll be able to more easily call into the newer interfaces from your code.

There is actually a change in the ADODataset code, a relevant revision (or bug fix, if you want to call it that way) of the ParseSQL of the TParameters used to extract parameters from an SQL statement.

dbExpress in Delphi 2010

In Delphi 2010 there are limited changes to the dbExpress core architecture, with most updates tied to specific drivers. This is due to the support for new databases (particularly Firebird, the open source spin-off of Interbase), new versions of existing databases, and changes in the way programs access the database (like in case of Microsoft SQL Server). The following sections have more details.

The Firebird Driver

Firebird is an open source database, developed by the Firebird Foundation, that originates from the version of Interbase that Borland released (even if only temporarily) as open source. Given the relationship with Interbase (the database distributed with Delphi since the early days of the product) and the fact that deployment of Firebird is free, it should come to no surprise that this database server is quite popular among Delphi developers. In the early days, the two database were highly compatible, and developers often used drivers and

components for Interbase to connect to Firebird, but over the years differences and incompatibilities started to emerge.

Despite this tight relationship, both Borland and the Firebird Foundation offered limited help in using the two together, because of previous discussions and misunderstandings among specific people on the different sides. It was only after the acquisition by Embarcadero Technologies that tensions cooled off, and the company started planning full support for both Interbase and Firebird in its database management line of tools and in its development tools.

For the first time since the Firebird project was started, Delphi 2010 has official and specific support for the database, in the form of a dbExpress driver. This is not like using the Interbase with different parameters, but it is, to all effects, a specific driver supporting the still popular Firebird 1.5 and the newer Firebird 2.1.1. The new driver has a specific DLL, relies on the Firebird client library (fbclient.dll) rather than the Interbase one (gds32.dll), although its entry point (*getSQLDriverINTERBASE*) is shared.

This is an example of the configuration of a *SQLConnection* component with the Firebird driver and one of the default databases:

```
object SQLConnection1: TSQLConnection
  ConnectionName = 'FBCONNECTION'
  DriverName = 'Firebird'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbxfb.dll'
  Params.Strings = (
    'drivername=Firebird'
    'database=local host: C:/Program Files/Firebird/
      Firebird2.1/examples/empbuid/employee.fdb'
    ...)
  VendorLib = 'fbclient.dll'
end
```

The unit you have to include in the project to manage the specific parameters is *DBXFirebird*. This unit is generally automatically included in forms and data modules with an *SQLConnection* component referring to the Firebird connection. In case you want to perform operations on metadata, the units to include start with the *DBXFirebirdMetaData* unit.

I haven't done extensive testing with using this driver against Firebird, but I've made a few experiments and it seems quite solid so far. The *DbxMulti2010* example already mentioned earlier for the themed *DBGrid* issues is in fact a Firebird application with the configuration listed above. I didn't have to make any other changes to convert it from Interbase to Firebird.


Updated dbExpress Drivers: Interbase, MySQL, Oracle

As in most new versions of Delphi, some of the drivers have been tested and updated to work with the latest version of the respective database server:

- The Interbase driver has been updated to Interbase 2009, including its To-Go version (the DLL-based version)
- The MySQL driver has been updated to MySQL 5.1 (notice that the MySQL client library, `libmysql.dll`, must match the server version or you'll see an exception)
- The Oracle driver has been updated to Oracle 11g.

The SQL Server Driver

The Microsoft SQL Server driver has been updated, specifically to support MS SQL Server 2008, but this also implies an architectural change. While in the past the driver relied on the OLDDDB driver (installed along with the MS Data Access Components, or MSDAC), the new driver uses the native client. The “Microsoft SQL Server 2008 Native Client” can be freely downloaded from the Microsoft web site as part of the “Microsoft SQL Server 2008 Feature Pack, August 2008” or individually in the same page that hosts the Feature pack (just scroll down to about half of the page). The URL⁸⁶ is:

 <http://www.microsoft.com/downloads/details.aspx?FamilyId=C6C3E9EF-BA29-4A43-8D69-A2BED18FE73C>

The architectural change in the dbExpress driver was required because the OLE DB support for SQL Server is frozen at SQL Server 2000 in terms of features, while the “native client” fully supports version 2005, version 2008, and future ones. This includes, for example, support for new data types such as date, time, `datetime2`, `datetimeoffset`, and the like.

In addition to the latest driver, with support for the SQL Server 2008 client library (`sqlncli10.dll`), there is a separate backward compatible dbExpress driver for SQL Server 2005. The key elements of the two configurations (extracted from the default `dbxdrivers.ini` file) are listed below:

⁸⁶ If you have to type it, you can also use the shorter version <http://bit.ly/5OuiEP>

```
[MSSQL] // Native Client for MS SQL Server 2008
LibraryName=dbxmss.dll
VendorLib=sqlncli10.dll

[MSSQL9] // SQL Native Client 2005
LibraryName=dbxmss9.dll
VendorLib=sqlncli.dll
```

Notice that the new dbExpress driver for MS SQL also includes support for Multiple Active Results Sets (also known by the MARS acronym).

DataSnap Updates

We have seen that dbExpress has seen enhancements in several drivers, but nothing has changed in terms of its architecture, the components you use, or their core features. After its debut in Delphi 2007, the dbExpress IV architecture is becoming quite stable.

This is not the case with DataSnap, Delphi's three-tier architecture, which has been extensively modified in Delphi 2009 and sees the completion of that project in Delphi 2010.

Overview of DataSnap in Delphi 2009

Originally based on a COM architecture, in Delphi 2009 the DataSnap framework was rewritten in terms of connectivity (now based on native sockets) and overall architecture, removing all dependencies from COM.

On the server side, in Delphi 2009 you could use three components⁸⁷:

- **DSServer**, the main server configuration component, which is needed to wire all the other DataSnap components together.
- **DSServerClass**, a component needed for each class you want to expose. This component is not the class you make available, but acts as a class factory to create objects of the class you want to call from a remote client. In other words, the DSServerClass component will refer to the class that has the public interface.

⁸⁷ This description is extracted from my “Delphi 2009 Handbook”, which has more details and specific examples I’m not going to repeat here.

- **DSTCPServerTransport**, a component that defines the transport protocol to be used (this is the only protocol directly available in Delphi 2009) and its configuration, such as which TCP/IP port to use.

On the client side, you still use the ClientDataSet component for caching the remote data, but the way you connect to the server has changed from the past. The components involved are:

- **SQLConnection**, generally used for dbExpress connections, has a DataSnap driver you can configure with the proper TCP/IP port.
- **DSPProviderConnection**, used to refer to the server class, with the ServerClassName property. This is the actual class you want to work with, not the DSServerClass object. This DSPProviderConnection can be referenced by the RemoteServer property of the ClientDataSet.
- **SqlServerMethod**, used to invoke a server side method directly (as if it was a stored procedure in a database).

Overview of DataSnap in Delphi 2010

Given this new foundation, in Delphi 2010 there have been several extensions. The most significant is probably the addition of a new connectivity option, HTTP, which can be used instead of sockets or alongside with them.

Along with HTTP support, which still relies on dbExpress for client connectivity, DataSnap in Delphi 2010 also has REST support, which lets you create clients in any language which can send an HTTP request and process the resulting JSON data structure. We'll explore REST support in the next chapter.

Beside HTTP and REST connectivity, new features of DataSnap include the filtering system, support for callbacks, and passing objects using the JSON marshaling layer. There are also a couple of very nice wizards to start the development of a new standalone DataSnap application or one hooked to an HTTP server (based on the classic Delphi WebBroker framework). Now that you should have an idea of the key elements, I can start getting to the actual details, starting with HTTP support.

DataSnap over HTTP

As a first example of DataSnap based on the new HTTP connectivity, I've extended the First3Tier2009 demo that I built for the Delphi 2009 Handbook⁸⁸. In the server I replaced the TCP/IP transport component with the DSHTTPService component, configured as:

```
object DSHTTPService1: TDSHTTPService
  RESTContext = 'rest'
  Server = DSServer1
  Filters = <>
  HttpPort = 8090
  Active = False
end
```

The `Active` property of the component is set to `True` when the server main form is created. That's the only change I had to make.

Notice I could have kept the TCP/IP connection along side the HTTP one: it might make sense to use the direct TCP/IP connection for internal clients running within the company (inside the firewall and in a protected Intranet or under a VPN), and open up the HTTP port for external connections coming from the Internet. I'll cover some more scenarios and configuration options later in this chapter.

For now, having migrated the server let me focus on the client. In this case, what we have to do is update the `SQLConnection` component configuration, which becomes the following:

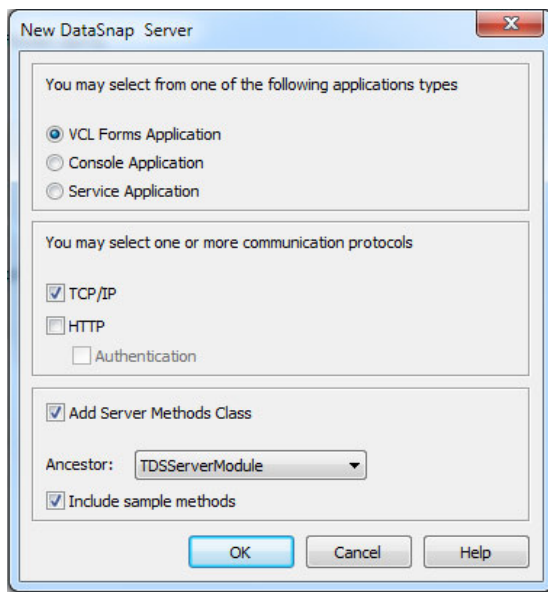
```
object SQLConnection1: TSQLConnection
  DriverName = 'Datasnap'
  LoginPrompt = False
  Params.Strings = (
    'Port=8090'
    'CommunicationProtocol=http'
    'DriverUnit=DBXDataSnap')
end
```

That's it. All I had to do was change the configuration of the DataSnap driver, typing *http* instead of *tcp/ip* and enter the correct port number, matching the one I'd configured on the server.

⁸⁸ Full source code of the example is available in the code section of my web site at:
http://www.marcocantu.com/code/dh2009/First3Tier2009_Server.htm and
http://www.marcocantu.com/code/dh2009/First3Tier2009_Client.htm.

A DataSnap HTTP Server with the Wizard

For that first demo I've taken an existing DataSnap application and moved it from a TCP/IP connection to an HTTP connection. It is even easier to create a new application with the DataSnap Wizard and ask for either or both connectivity options. If you select the DataSnap Server in the DataSnap page of the New Item dialog box (or Object Repository) you'll get the following options:



You can pick three different application architectures, one or both communication protocols (optionally asking for HTTP Authentication) and add a ready to use server method class based on a DataSnap Server Module, a plain Data Module, or a plain TPersistent ancestor. Depending on the options you pick, you can have a variety of structures for your DataSnap server. In each case, the wizard will generate a data module with the core DataSnap server components, plus the modules you ask for.

Lets suppose I pick a Console Application, HTTP with Authentication, and a server method class based on a TPersistent ancestor with the sample methods. The wizard will generate two units and a project file, available in the DSnapHttpConsole project with the original unit names suggested by the DataSnap Server Wizard.

The main project file for the console application, has a single call (protected by an exception handler) to a RunDSServer function:

```
try
  RunDSServer;
except
  on E: Exception do
    Writeln(E.ClassName, ': ', E.Message);
end
```

The ServerContainerUnit1 unit is a data module hosting the DSServer, DSServerClass, and DSHTTPService components already used in the previous example, plus the DSHTTPServiceAuthenticationManager component I asked for in the Wizard.

```
object ServerContainer1: TServerContainer1
  object DSServer1: TDSServer
    AutoStart = True
    HideSAdmin = False
  end
  object DSHTTPService1: TDSHTTPService
    Server = DSServer1
    DSHostname = 'localhost'
    AuthenticationManager =
      DSHTTPServiceAuthenticationManager1
    HttpPort = 8090
  end
  object DSHTTPServiceAuthenticationManager1:
    TDSHTTPServiceAuthenticationManager
  end
  object DSServerClass1: TDSServerClass
    OnGetClass = DSServerClass1GetClass
    Server = DSServer1
    LifeCycle = 'Session'
  end
end
```

Beside the port and host name configuration, notice the OnGetClass event handler of the DSServerClass is defined there and implemented by the Wizard.

In the server class configuration above, I let the DSServerClass component use the default value of the LifeCycle property, *Session*, but this is totally ignored and behaves like a TCP/IP server when the *Invocation* life cycle is used. This means a new server class object is created for each client request, which should not be surprising given the use of HTTP. The HTTP connection is a stateless connection, which means a new connection to the server is established for each client request (with some very limited exceptions).

In case of a console application, the ServerContainerUnit1 unit also implements the RunDSServer global function, which creates the data module and starts the

DSServer. The code waits until the *Esc* key is pressed to terminate the console application:

```
var
  LModule: TServerContainer1;
  LInputRecord: TInputRecord;
begin
  LModule := TServerContainer1.Create(nil);
  try
    LModule.DSServer1.Start;
    try
      while True do
        begin
          ... check for Esc key
```

The ServerMethodsUnit1 unit has the target class (automatically connected to the DSServerClass component of the main unit, as mentioned earlier) with the sample method:

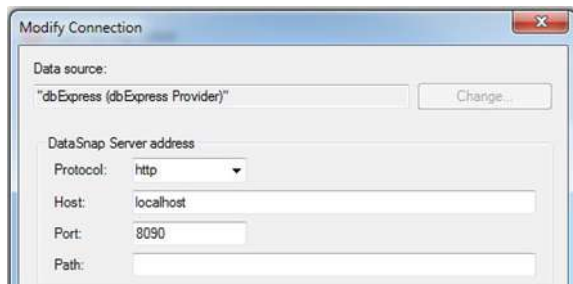
```
type
{$METHODINFO ON}
  TServerMethods1 = class(TPersistent)
  public
    function EchoString(Value: string): string;
  end;
{$METHODINFO OFF}
```

This is all the code generated for the server. We can try to compile and run it as it is, but how do we test it without creating a client application? Turns out we can use the Data Explorer pane of the Delphi IDE.

Testing the Connection in Data Explorer

It is interesting to notice that in the Data Explorer window of the Delphi 2010 IDE you can configure and test the client connection quite easily. It is actually easier to test the connection in this pane than setting up an SQLConnection component manually.

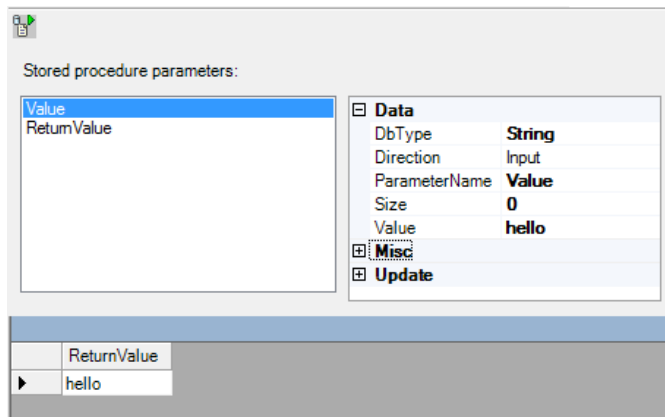
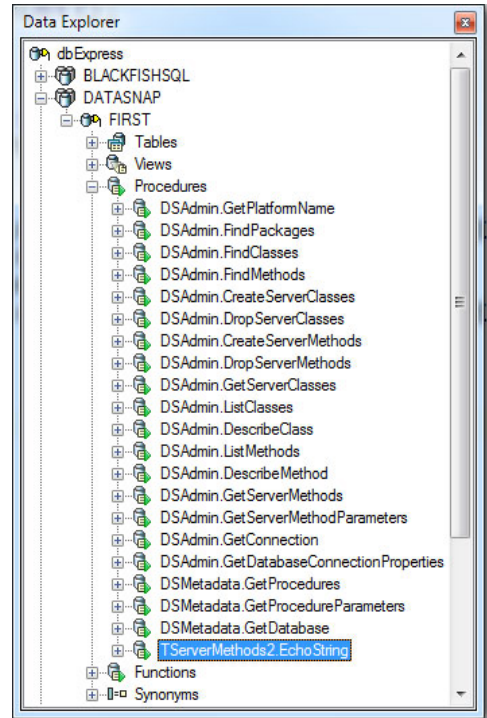
To try this out, simply run the server (possibly as a stand alone application using the Run Without Debugging command), move to the Data Explorer, pick the DataSnap driver, and add a new connection. Now open the



connection configuration and enter the settings for your server (including host and port), as shown here. Test the connection to see if it is OK and the server is running.

Now you can check the connection features in the Data Explorer pane. The DataSnap connection will have no tables, but will have a set of procedures corresponding to the methods exposed by the server, which include the administrative methods and the specific ones provided by your server class (or classes). With this example you'll see the list on the side of this page.

Now you can even select the given server method, open its parameters, set their Value property (in this case the parameter itself is called Value, causing some confusion), and even execute it from the IDE by using the right mouse button in the pane with the *stored procedure* parameters, displayed below. Here the result data set, which has a single element with the return value (the parameter echoed from the server back to the client), is displayed.

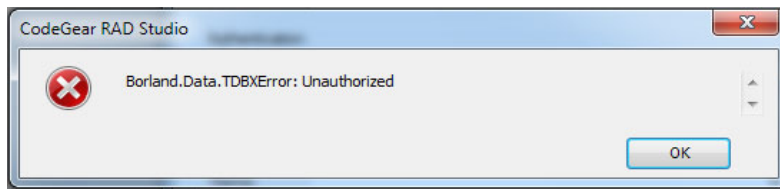


HTTP Authentication

In this application I asked the Wizard to add support for HTTP authentication, but haven't actually used it so far. What you have to do is handle the `HTTPAuthenticate` event of the `DSHTTPServiceAuthenticationManager` component. In this event handler you receive the user name and the password, but also more information about the incoming request (so you could have a different authentication strategy depending on the request). In this very simple case I've used the simplest possible implementation, a fixed user name and password (no, it is not my real password!):

```
procedure TServerContainer1.  
  DSHTTPServiceAuthenticationManager1HTTPAuthenticate(  
    Sender: TObject;  
    const Protocol, Context, User, Password: string;  
    var valid: Boolean);  
begin  
  valid := (User = 'marco') and (password = '123');  
end;
```

Notice that the authentication request comes in for every single HTTP request, as the protocol is inherently stateless. Now if you compile and run the server with authentication support and connect from the Data Explorer DataSnap driver, you'll see an error like:



All you have to do is enter the user name and password in the Authentication section of the connection configuration (displayed earlier in the section “Testing the Connection in Data Explorer”).

Building a Client Application the RAD Way

There isn't a wizard to build DataSnap client applications, but once you have configured the connection in Data Explorer, you can take advantage of its drag-and-drop capabilities.

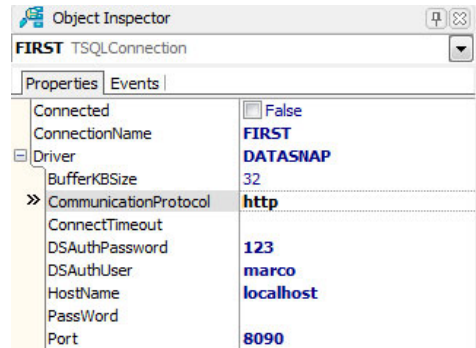
For example, if you drag the connection defined earlier and called “FIRST” over the form, you'll get the following:

```

object FIRST: TSQLConnection
  ConnectionName = 'FIRST'
  DriverName = 'DATASNAP'
  LoginPrompt = False
  Params.Strings = (
    'drivername=DATASNAP'
    'port=8090'
    'communicationprotocol=http'
    'hostname=localhost'
    'DSAuthenticationUser=marco'
    'DSAuthenticationPassword=123')
end

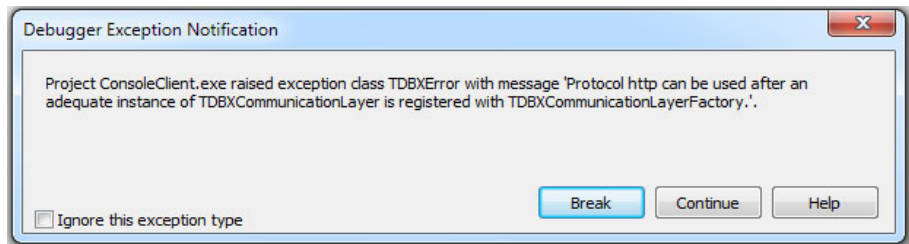
```

The interesting thing is that the configuration of the communication protocol is easier in the connection editor than it is in the Object Inspector. In the former, in fact, you have the various options (*http* or *tcp/ip*) in a combo box, while in the Object Inspector you have to type in the value, as you can see here on the right.



Notice also that the properties used for the user name and the password: these are not the Username and Password properties, but the *DSAuthUser* property (stored in the Params as *DSAuthenticationUser*) and the *DSAuthPassword* property (stored as *DSAuthenticationPassword*).

Once you have the SQLConnection component in place, we can think of turning it on. This won't work at design time, nor if you call the Open method of the connection at run time. In this second case you'll get the error:



The error indicates that the communication protocol has not been registered, and you can do this by adding a reference to the DSHTTPLayer unit:

```

uses
  DSHTTPLayer;

```

Now we can do another drag-and-drop operation for calling the only method of the server. If you open the list of procedures and drag the one you want to call, the forms will have an `SQLDataSet` component configured as a stored procedure:

```
object TServerMethods1_EchoString: TSQLDataSet
  CommandText = 'TServerMethods1.EchoString'
  CommandType = ctStoredProc
  SQLConnection = FIRST
end
```

If you try to call it, however, it will fail, as the remote server won't recognize a request referring to a stored procedure. What you have to do is manually change the `CommandType` to `ctServerMethod`. When you do so, however, the value of the `CommandText` property (which is correct) will be reset. So you should copy the `CommandText`, change the `CommandType`, and then paste the `CommandText` back⁸⁹.

Overall, you'll get the component configured as follows, including its input and output parameters:

```
object TServerMethods1_EchoString: TSQLDataSet
  CommandText = 'TServerMethods1.EchoString'
  CommandType = ctServerMethod
  DbxCommandType = 'DataSnap.ServerMethod'
  MaxBlobSize = -1
  Params = <
    item
      DataType = ftWideString
      Precision = 2000
      Name = 'Value'
      ParamType = ptInput
    end
    item
      DataType = ftWideString
      Precision = 2000
      Name = 'ReturnParameter'
      ParamType = ptResult
      Size = 2000
    end>
  SQLConnection = FIRST
end
```

Now all you have to write is the code for setting the input parameter, execute the server method, and retrieve the result:

⁸⁹ The fact that the text of the command is lost as you change the command types makes sense when changing from a table to a query, but when switching between two named objects (like in this case) it could have been preserved. I won't say this is a bug, though: better fix the drag-and-drop behavior!

```

procedure TConsoleClientForm.btnCallEchoClick(Sender: TObject);
begin
    TServerMethods1_EchoString.ParamByName('Value').
        AsString := 'Hello ' + TimeToStr (Now);

    TServerMethods1_EchoString.ExecSQL;

    ShowMessage (TServerMethods1_EchoString.
        ParamByName('ReturnParameter').AsString);
end;

```

As an alternative, you can avoid dragging the procedure and adding the data set altogether, by using the “Generate DataSnap client classes” command of the SQLConnection component to generate the client proxy classes for the server. I did that and saved the new unit as ClientProxy. Now you can refer to the unit and replace the component and the call above with the creation of an instance of the proxy and its use:

```

uses
    ClientProxy;

procedure TConsoleClientForm.btnProxyClick(Sender: TObject);
begin
    with TServerMethods1Client.Create(FIRST.DBXConnection) do
        try
            ShowMessage (EchoString('Hello ' + TimeToStr (Now)));
        finally
            Free;
        end;
    end;

```

This last part of the application was not exactly based on a visual development style, but it was certainly worth mentioning it as a relevant alternative for calling server methods.

DataSnap WebBroker Integration

Building an application to run on the server (whether a standalone VCL program, a service, or a console application) is only one of the options for deploying DataSnap servers in Delphi 2010 or, to be more precise, to offer HTTP connectivity to DataSnap servers.

The alternative option is to create and deploy Web server extensions, based on Delphi's classic WebBroker architecture.

Overview of the WebBroker Architecture

This overview is meant as a short introduction of a technology that has existed since Delphi 3⁹⁰, for those who never used it or used it only occasionally. If you already used WebBroker you can certainly skip it.

The WebBroker technology, available in Delphi since the early days of the product, is a framework to let you create Web server extensions that can be deployed as CGI applications, ISAPI libraries, and (even if unofficially) Apache modules. There is a fourth option, which is the use of a debug tool, called Web App Debugger as a replacement for a web server while developing and debugging the application.

A WebBroker application is built around a WebModule designer, which has an object holding the web request received from the client and the web response, plus a collection of actions tied to the incoming URLs. This TWebModule derives from TCustomWebDispatcher, which provides support for all the input and output of your programs and defines the Request and Response properties.

These properties are defined using a base abstract class (TWebRequest and TWebResponse), but an application initializes them using a specific object (such as the TISAPIRequest and TISAPIResponse subclasses for an ISAPI library). These classes make available all the information passed to the server, so you have a single approach to accessing all the information.

The key advantage of this approach is that the code written with WebBroker is independent of the type of application (CGI, ISAPI, Apache module); you'll be able to move from one to the other, modifying the project file or switching to another one, but you won't need to modify the code written in a WebModule.

To write the application code, you can use the Actions editor in the WebModule to define a series of actions (stored in the Actions array property) depending on the *path name* of the request. This path name is the portion of request URL that comes after the program name and before the parameters.

By providing different actions, your application can easily respond to requests with different path names, and you can assign a different producer component or call a different OnAction event handler for each and every possible path

90 As a historical reference you can see <http://edn.embarcadero.com/article/10134>

name. In the `OnAction` event handler, you write the code to specify the response to a given request, as in the following code snippet, which returns some plain HTML:

```
procedure TWebModule1.WebModule1WebActionItem1Action (
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<html><head><title>Hello Page</title></head><body>' +
    '<h1>Hello</h1>' +
    '<hr><p><i>Page generated by Marco</i></p>' +
    '</body></html>';
end;
```

As debugging web applications is often difficult, Delphi offers a specific Web App Debugger program. This program, which is activated by the corresponding item on the Tools menu, is a web server that waits for requests on a port you can set up (8081 by default). When a request arrives, the program can forward it to a stand-alone executable using a socket connection.

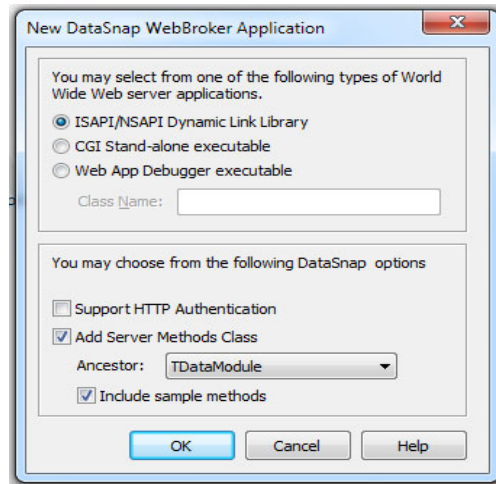
Using this tool you can run the web server application from within the Delphi IDE, set all the breakpoints you need, and then debug the program as you would a plain executable file.

The Web App Debugger also does a good job of logging all the received requests and the responses returned to the browser, letting you inspect the data flow at the HTTP protocol level.

The DataSnap WebBroker Wizard

Creating DataSnap applications based on the WebBroker architecture is reasonably simple thanks to another new Wizard added to the Delphi 2010 IDE, the DataSnap WebBroker Application wizard.

Differently from the DataSnap Application Wizard, this one won't let you pick a TCP/IP connection and offers you the application types supported by WebBroker as options, rather than the development of full-blown, console, or service applications. You can see the Wizard form in the next page.



To build an example I've picked the Web App Debugger executable option and given a name to its *class*⁹¹. The Wizard generates a unit with the target Server Methods class (again a DataSnap Module, a Data Module or a TPersistent descendant), plus a Web Module hosting the DSServer and DSServerClass components.

The web module has one extra new component, DSHTTPWebDispatcher, which provides the conduit between WebBroker and DataSnap, by hooking to the web dispatch mechanism and intercepting the incoming requests that start with a given path name. The component has these default settings:

```
object DSHTTPWebDispatcher1: TDSHTTPWebDispatcher
  RESTContext = 'rest'
  Server = DSServer1
  DSHostname = 'localhost'
  DSPort = 211
  WebDispatcher.MethodType = mtAny
  WebDispatcher.PathInfo = 'datasnap*'
end
```

The key property here is the PathInfo intercepted by this dispatcher, anything starting with *datasnap*. The REST information is relevant, but I'll focus on that topic in the next chapter. Finally, the DataSnap host and port configurations are used only in case of a gateway, covered in a later section.

91 This *class name* is a registration name (nothing to do with a Delphi class) provided by the program and used along with the application name to refer to the module in a URL. This name is saved in a registration line inside the initialization section main form unit, in a call to the constructor of the TWebAppSocketObjectFactory class.

Using this Wizard I've built the DSnapWebAppDebug project, which registers the *'dsnapihttp'* class name (we'll see its URL as we move to the client-side project). The project has an empty and useless main form, like any Web App Debugger application, a web module, and a data module.

The web module has a DSHTTPWebDispatcher component with the default settings I've just listed, a DSServer component and a DSServerClass, connected to the sample target data module. The web module also has a default action (that is an action matching any path not configured in a dispatcher) defined as:

```
object WebModule2: TWebModule2
  Actions = <
    item
      Default = True
      Name = 'DefaultHandler'
      PathInfo = '/'
      OnAction = WebModule2DefaultHandlerAction
    end>
```

The code of this event handler is quite trivial:

```
procedure TWebModule2.WebModule2DefaultHandlerAction (
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<html><heading><body>' +
    'DataSnap Server</body></html>';
end;
```

Rather than changing this HTML⁹² in code, you can return the content of a local HTML file, in my code *'main.html'*, to be used as a standard response. This way you can customize the default page returned by the server without having to recompile the application. This is the updated OnAction event handler for the default action:

```
var
  strRead: TStreamReader;
begin
  strRead := TStreamReader.Create('main.html');
  Response.Content := strRead.ReadToEnd;
  strRead.Free;
```

You can add actions to this web module, like any WebBroker application.

The third unit of the project is a data module, which has a Firebird dbExpress connection (but you can use any other supported target database). The three

92 Oddly enough the code generated by the Wizard uses a *heading* tag rather than HTML's *head* tag. Not a big deal, as you are supposed to replace it with your own HTML anyway.

components in this data module are (a connection, a data set, and a provider), listed here with only their key properties:

```

object FBCONNECTI ON: TSQLConnecti on
    Connecti onName = 'FBCONNECTI ON'
    Dri verName = 'Fi rebird'
end
object CUSTOMER: TSQLDataSe t
    CommandText = 'CUSTOMER'
    CommandType = ctTable
    SQLConnecti on = FBCONNECTI ON
end
object DataSe tProvi der1: TDataSe tProvi der
    DataSe t = CUSTOMER
end

```

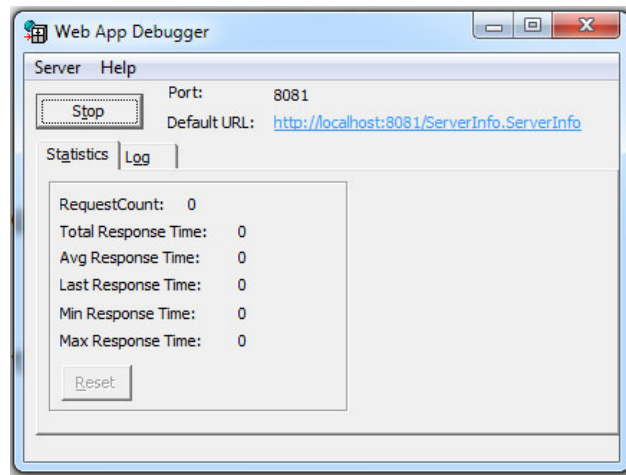
The program has also two methods, the sample EchoStri ng (which in this case doubles the string passed as parameter) and a GetDataSe t method returning a data set:

```

publ i c
    functi on EchoStri ng(Value: string): string;
    functi on GetDataSe t: TDataSe t;

```

Once you compile this application, you not only have to run it but also activate the Web App Debugger (from the Tools menu of the Delphi IDE), which acts as the web server which will redirect the calls to the application.



Once you open the Web App Debugger you can start its service and it will suggest the URL of the ServerInfo application, which will list all programs that you've registered for the Web App Debugger. This lets you easily find the URL

of the given project. However, if you know how this is defined, it won't be difficult to get it manually:

```
http://localhost:8081/dsnapwebappdebug.dsnap1http
```

First comes the server and the port, followed by the application name *dot* the registered class name I mentioned earlier. If you type this URL in a browser you'll see the static HTML file returned by the program.

Although you could possibly try to invoke specific features of the server, there is very little you can do with the HTTP interface, which remains basically a proprietary interface. To be able to refer to the server methods from a browser you'll need to use the new REST support of DataSnap, covered in the next chapter.

A Client for the Web Server

Now that we have built a DataSnap server deployed as a web server extension, we have to change a few things in how a client application refers to it. The SQL-Connection component, in fact, won't refer to the server using host name and port, but using the server base URL:

```
object SQLConnection1: TSQLConnection
  DriverName = 'Datsnap'
  LoginPrompt = False
  Params.Strings = (
    'URLPath=http://localhost:8081/' +
    'DSnapWebAppDebug.dsnap1http'
    'CommunicationProtocol=http')
end
```

Again, you'll also need to add the DSHTTPLayer unit to the client application. This time beside calling the EchoString method on the server, the client application reads the remote data set into a local ClientDataSet component, showing the data in a DBGrid. Nothing new for those who have used DataSnap in Delphi 2009, but slightly different from past versions of this technology.

The key components I've added to the main form of the client application to fetch the data set from the server are a SQLConnection, a DSProviderConnection referring to the server class, and a ClientDataSet pointing to a specific provider:

```
object SQLConnection1: TSQLConnection
  DriverName = 'Datsnap'
  Params.Strings = (
    'URLPath=http://localhost:8081/' +
```

```

        'DSnapWebAppDebug.dsnap1http'
        'CommunicationProtocol=http')
    end
    object DSProviderConnection1: TDSProviderConnection
        ServerClassName = 'TServerMethods1'
        SQLConnection = SQLConnection1
    end
    object ClientDataSet1: TClientDataSet
        ProviderName = 'DataSetProvider1'
        RemoteServer = DSProviderConnection1
    end
    object DataSource1: TDataSource
        DataSet = ClientDataSet1
    end
    object DBGrid1: TDBGrid
        DataSource = DataSource1
    end
end

```

To fetch the data from the server there is no specific code, but a request to open the ClientDataSet as the program starts. The client application has some actual code used to call the Server Methods, but this is exactly like the earlier code.

Overall, we have deployed our DataSnap server as a Web Server extension, and after the development could move to a CGI, ISAPI DLL⁹³, or Apache Module project. Still, the client is nothing but a Delphi client which uses the DataSnap driver of dbExpress, much like the earlier HTTP DataSnap servers or the TCP/IP based ones.

Filtering Connections

One of the most relevant requests developers had for DataSnap was the ability to compress and encode the data stream. The former helps with performance, while the second makes sniffing the data moved over the wire a little more difficult. In Delphi 2010 DataSnap introduces more than this, as it features a filtering architecture you can use to hook any filter to the input and output streams. Even more, the server publishes information about the filters it used, so that the client can decode the stream in a proper way (but can also connect to multiple different servers with the same basic code).

93 For an example of deployment of a DataSnap WebBroker application in IIS, you can refer to the following blog entry by Delphi R&D member Jim Tierney: <http://blogs.embarcadero.com/jimtierney/2009/08/20/31502>

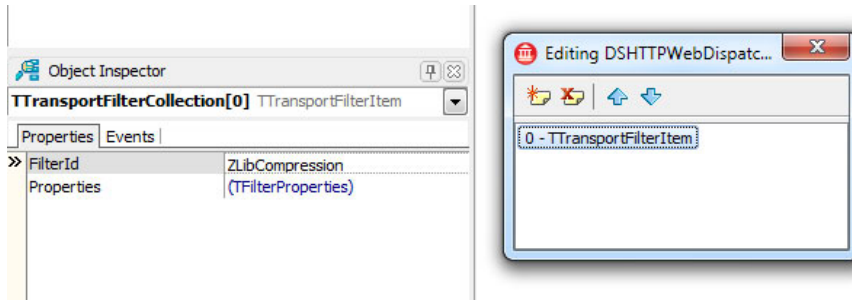
Using ZlibCompression

Let me start by showing you how to use the only filter⁹⁴ the ships with Delphi, the compression filter. First, I've built a sample DataSnap WebBroker server, using the Web App Debugger support, and built a corresponding client. They can be found in the DSnapFilterDemo folder. The reason for picking the Web App Debugger is you can easily monitor the HTTP data. For example, the response of the method call passing the string *'This is my name'* looks like:

```
HTTP/1.1 200 200 OK
Connection: close
Content-Type: text/html
Content-Length: 57
```

```
{"result": [{"rows": [0]}, {"data": [17, Å/This is my name]}]}
```

As a second step, I've added the ZLibCompression transport filter to the server side application (more precisely to the DSHTTPWebDispatcher component of the DataSnap server). Just edit the Filters collection of this component, add an entry, and configure it in the Object Inspector:



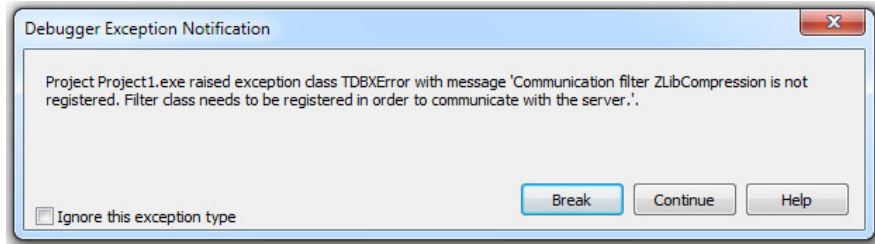
The DSHTTPWebDispatcher component will have the following configuration:

```
object DSHTTPWebDispatcher1: TDSHTTPWebDispatcher
  RESTContext = 'rest'
  Server = DSServer1
  Filters = <
    item
      FilterId = 'ZLibCompression'
    end>
  WebDispatcher.MethodType = mtAny
  WebDispatcher.PathInfo = 'datasnap*'
end
```

94 Rumors say that more filters were ready to ship, but Embarcadero's legal department got worried about shipping strong encryption software, so they were removed at the very last minute.

230 - Chapter 7: Database Access and DataSnap

Remember that you can also apply filters to a DSTCPServerTransport component, in the case of a socket-based DataSnap server. Now if you fire up the client application, you'll see an error like the following:



As the error message indicates, the exception is due to the fact that the client request doesn't recognize the filter, which was not compiled and registered into the client application.

What you need to do is to add the compression filter unit to the client program (beside the DataSnap HTTP client support unit every DataSnap client based on HTTP must declare):

uses

DSHttpLayer, DbxCompressionFilter;

Now the client program works again, with the compressed stream. The HTTP data for the simple response becomes the following unreadable text:

```
HTTP/1.1 200 200 OK
Connection: close
Content-Type: text/html
Content-Length: 61

x#«V*J-. í)QzŠ®V*Ê//#2
bkuª•R#K##CS#ú! #™Ä
@"[©-~>#[#[#
```

There is actually no real compression in this very short response (the Content-Length is about the same), but there is when the amount of data is bigger. Now this data looks unreadable, but a simple Unzip request will reveal the content.

What we might need is to encode or encrypt the content. How do we accomplish this? In more general terms, if ZLibCompression is the only available DataSnap filter, how do we implement a custom one?

Creating Custom Filters

To add a custom filter to the DataSnap connection layer, what you have to do is:

- Create a package featuring a *transport filter* class that inherits from the `TTransportFilter` class
- Implement the `ProcessInput` and `ProcessOutput` virtual methods of this class
- Register the transport filter and install the package with the filter in the Delphi IDE.

The `DSnapFilterDemo` folder contains the `DSnapFilter01` package with a trivial MIME encoding filter. The class of the filter is defined as:

```
uses
  DBXTransport, IdCoderMIME;

type
  TMimeFilter = class (TTransportFilter)
  public
    function ProcessInput(const Data: TBytes): TBytes; override;
    function ProcessOutput(const Data: TBytes): TBytes; override;
    function Id: string; override;
  end;
```

The code of these three methods is not particularly complex. The `Id` function returns a unique identifier for the filter (passed also to the client in the stream):

```
function TMimeFilter.Id: string;
begin
  Result := 'Cantools.MimeFilter';
end;
```

The `ProcessInput` and `ProcessOutput` methods process the bytes stream using Indy's `TIdEncoderMIME` and `TIdDecoderMIME` support classes:

```
function TMimeFilter.ProcessInput(const Data: TBytes): TBytes;
var
  strEncoded: string;
begin
  strEncoded := TIdEncoderMIME.EncodeBytes(Data);
  Result := BytesOf(strEncoded);
end;

function TMimeFilter.ProcessOutput(const Data: TBytes): TBytes;
var
  strEncoded: string;
begin
  strEncoded := StringOf (Data);
  Result := TIdDecoderMIME.DecodeBytes(strEncoded);
end;
```


The unit also performs the registration (and de-registration) of the filter, something both the server and the client must do.

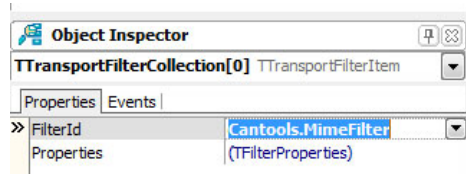
Initialization

```
TTransportFilterFactory.RegisterFilter(TMimeFilter);
```

Finalization

```
TTransportFilterFactory.UnregisterFilter(TMimeFilter);
```

Also the package needs the registration code to let you see the new transport filter at design time and pick it from the list of filters in the Object Inspector (displayed here on the right side) while configuring the Filters collection of the DSHTTPWebDispatcher component. The settings of this component now



become the following (this is the code in the final version of the demo):

```
object DSHTTPWebDispatcher1: TDSHTTPWebDispatcher
  RESTContext = 'rest'
  Server = DSServer1
  Filters = <
    item
      FilterId = 'Cantools.MimeFilter'
    end>
  WebDispatcher.MethodType = mtAny
  WebDispatcher.PathInfo = 'datasnap*'
end
```

Notice that you can have multiple filter processing the data stream. The client will apply them in the reverse sequence of the server. Keep also in mind that in this case both the client and the server applications must include the filter unit, which in the example is the DSnapFilter64 unit. With this specific filter, the HTTP data packet when receiving the response of the server method becomes:

```
HTTP/1.1 200 200 OK
Connection: close
Content-Type: text/html
Content-Length: 76

eyJyZXN1bHQiOiB7IjEvd3MiOiBwXX0seyJkYXRhIjpbMTcswC9UaGlzIGlziG15IG5hbWVdfV19
```

A MIME-encoded data stream provides just a little bit of data garbling, but it isn't in any way a secure mechanism. If you are interested in real encryption filters for DataSnap, you can refer to the *hash* and *cipher* filters part of the “DataSnap Filters Compendium” that Daniele Teti put together and you can find on Google Code at:

<http://code.google.com/p/dsfc/>

JSON and Object Marshaling

There is another new feature in Delphi 2010 that makes a huge difference in the architecture: the types of parameters for server methods now include `TJSONValue` and descendant types. What is relevant here is not the ability to pass individual values (integers, strings), but the fact you can use this approach to pass complex data structures.

While DataSnap still doesn't allow you to pass Delphi objects to and from server methods, what you can do is transform Delphi objects into an equivalent JSON representation and then back into actual objects. This technique is called *marshaling* and is worth investigating in detail. But before we get to that, let me start covering one of the foundations, that is the JSON representation.

Introducing JSON

The acronym JSON stands for JavaScript Object Notation. This is a text-based notation used to represent JavaScript objects⁹⁵, so that they can be made persistent or transferred from one application to another (or from one computer to another). While a few years ago the consensus was for using XML as a notation to represent complex data structures, in the last few years JSON has gained popularity because it is more compact, more tied to programming language concepts, and very easy to parse in JavaScript browser based applications... and by most other programming languages due to a growing set of libraries and tools.

You can learn more about JSON by reading the RFC 4627 specification of the IETF (Internet Engineering Task Force) or looking at the *official* JSON home page:

```
http://www.ietf.org/rfc/rfc4627.txt
http://json.org
```

⁹⁵ More precisely, it is a subset of JavaScript's object literal notation. In JavaScript, the use of double quotes around the pair name is not required for valid variable names.

You can also keep reading for a short introduction, as JSON is relatively simple to understand, with 4 primitive types and two structures. The primitive types are numbers, strings, Booleans⁹⁶ (true or false) and the null value.

The two JSON data structures are:

- JSON Objects - Collections of name and value pairs, enclosed in curly braces and separated by commas (while the two elements of each pair are divided by a colon); collections represent records or objects
- JSON Arrays - Lists of values within square brackets and separated by commas; lists represent arrays or collections

Here are simple examples of the two notations, an object with two *pairs* (all strings including pair names are indicated by double quotes) and a list of two values, a number and a string:

```
{
  "Name": "Marco",
  "Value": 100
}

[22, "foo"]
```

Of course, you can combine these constructs at will, so you can use objects and arrays as values of a pair and as elements of an array:

```
{
  "Me": {
    "FirstName": "Marco",
    "LastName": "Cantù",
    "Wife": "Lella",
    "Kids": [{"Name": "Benedetta", "Age": 10},
             {"Name": "Jacopo", "Age": 6}]
  }
}
```

JSON in Delphi 2010

While there have been a few JSON libraries for Delphi in the past, the first edition with native support is Delphi 2010. The native JSON support has been made available through a series of classes defined in the DBXJSON unit, which (despite the name) can be used even in applications that don't relate to the dbExpress framework.

96 As we'll see later, Delphi treats JSON Boolean values as two special values: true and false.

The DBXJSON unit defines classes that you can use to work with the various JSON data types (individual values of different types, arrays, pairs, and objects) all inheriting from `TJSONValue`:

- Primitive values include `TJSONNull`, `TJSONFalse`, `TJSONTrue`, `TJSONString` and `TJSONNumber`.
- Data structures include `TJSONObject` (and the internal `TJSONPair`) and `TJSONArray`.

Here is a simple code fragment, extracted from the `JsonTests` project, used to demonstrate the output of the different primitive types. Notice that each temporary object you create must be manually freed, hence the idea of adding the `LogAndFree` private support method:

```
procedure TFormJson.LogAndFree (jValue: TJSONValue);
begin
  try
    Log (jValue.ClassName + ' > ' + jValue.ToString);
  finally
    jValue.Free;
  end;
end;

procedure TFormJson.btnValuesClick(Sender: TObject);
begin
  LogAndFree (TJSONNumber.Create(22));
  LogAndFree (TJSONString.Create('sample text'));
  LogAndFree (TJSONTrue.Create);
  LogAndFree (TJSONFalse.Create);
  LogAndFree (TJSONNull.Create);
end;
```

This is the corresponding output:

```
TJSONNumber > 22
TJSONString > "sample text"
TJSONTrue > true
TJSONFalse > false
TJSONNull > null
```

It is equally simple to use the other classes of the DBXJSON unit for creating arrays and objects. An array is a structure to which you can add any value (including arrays and objects):

```
procedure TFormJson.btnSimpleArrayClick(Sender: TObject);
var
  jList: TJSONArray;
begin
  jList := TJSONArray.Create;
  jList.Add(22);
  jList.Add('foo');
  jList.Add(TJSONArray.Create (TJSONTrue.Create));
```

```

    (jList.Get (2) as TJsonArray).Add (100);
    Log (jList.ToString);
    jList.Free;
end;

```

The JSON output shows the two nested arrays as follows:

```
[22, "foo", [true, 100]]
```

Note that the JSON containers (arrays and objects) own their internal elements, so that you can free the container to clean up the memory for the entire group of JSON values.

When you have an object, the only element you can add to it is a pair, but the value of the pair can be just any JSON value, including a nested object:

```

procedure TFormJson.btnSimpleObjectClick(Sender: TObject);
var
    jsonObj, subObject: TJSONObject;
begin
    jsonObj := TJSONObject.Create;
    jsonObj.AddPair(TJSONPair.Create ('Name', 'Marco'));
    jsonObj.AddPair(TJSONPair.Create ('Value',
        TJSONNumber.Create(100)));

    subObject := TJSONObject.Create(
        TJSONPair.Create ('Subvalue', 'one'));
    jsonObj.AddPair(TJSONPair.Create ('Object', subObject));

    Log (jsonObj.ToString);
    jsonObj.Free;
end;

```

The JSON representation of this object (with a little manual formatting to improve readability) is the following:

```

{
  "Name": "Marco",
  "Value": 100,
  "Object": {
    "Subvalue": "one"
  }
}

```

Parsing JSON

Creating JSON data structures using the DBXJSON classes and generating the corresponding JSON representation is interesting, but it is even more interesting to know you can do the reverse, that is parse a string with a JSON representation to create the corresponding in memory objects.

Once you have a JSON string, you can pass it to the `ParseJSONValue` class method of the `TJSONObject`, which returns a `TJSONValue` object⁹⁷. In cases where we know the evaluation returns a JSON object we have to cast it back to the proper type. The `ParseJSONValue` class method doesn't accept a string as parameter, but requires an array of bytes with an ASCII encoding. So we need to take the string and encode it using the `TEncoding` class, that is by calling `TEncoding.ASCII.GetBytes`.

Overall, the initial portion of the `btnParseObjectClick` (still part of the main form of the `JsonTests` example) is the following:

```
var
  strParam: string;
  jsonObj: TJSONObject;
begin
  strParam := '{"value": 3}';
  jsonObj := TJSONObject.ParseJSONValue(
    TEncoding.ASCII.GetBytes(strParam), 0) as TJSONObject;
```

The remaining code outputs the entire object (getting back the original JSON representation, if nothing went wrong), the last (and only) name/value pair and frees the `TJSONObject` object (again, it is easy to cause memory leaks with this type of code):

```
Log (jsonObj.ToString);
Log (jsonObj.Get (jsonObj.Size - 1).ToString);
jsonObj.Free;
```

Streaming Objects to JSON

Now if it is this easy to create a `TJSONObject` and add data to it, it would be very nice to create the JSON representation of any Delphi object. This is possible in Delphi 2010 thanks to the JSON marshaling and de-marshaling support that is defined in the `DBXJSONReflect` unit.

This unit defines an open and extensible marshaling architecture, and provides a specific implementation based on the internal `TJSONConverter` class. So, by default, you create a JSON marshaller by passing the JSON converted to it:

```
jMarshal := TJSONMarshal.Create(TJSONConverter.Create);
```

97 If you find it confusing that you have to use a `TJSONObject` class method to parse any JSON value and return a `TJSONValue` (which might be an object, an array, a primitive value) you are not alone. It seems quite a random pick to me. I think I'd preferred a global function or a class method of the `TJSONValue` class.

The marshaling engine will own the converter and free it when it is done by default. You can register more converters to support different formats, including XML and others. The engine behind the marshaling support is the new Extended RTTI, which I covered in detail in Chapter 3. RTTI is used for saving an image of the objects (including all of the fields) and for re-creating a type given its name. The overall implementation is relatively complex and I don't really want to delve into the details, but only focus on a practical example.

Let's suppose we have a class like this one defined in the `JsonMarshal` project (notice I'm using private data and no published properties and the class is not a `TPersistent` class):

```
type
  TMyData = class
  private
    theName: String;
    theValue: Integer;
  public
    constructor Create (const aName: string);
    function ToString: string; override;
    property Value: Integer read theValue write theValue;
  end;
```

We can now write some code to create an object of this class, create the marshaller, marshal it to a `TJSONValue`, and add the result to a Memo control:

```
theData := TMyData.Create('john');
theData.Value := 99;
jMarshal := TJJSONMarshal.Create(TJSONConverter.Create);
jValue := jMarshal.Marshal(theData);
Memo1.Lines.Text := jValue.ToString;
```

In the actual code this takes place within three nested try-finally blocks (omitted here), each freeing the object created in the given step in reverse order:

```
jValue.Free;
jMarshal.Free;
theData.Free;
```

At the end of the operation the Memo control will have the JSON representation of the object created by converting the original Delphi object into its `TJSONObject` equivalent:

```
{
  "type": "JsonMarshal_MainForm.TMyData",
  "id": 1,
  "fields": {
    "theName": "john",
    "theValue": 99
  }
}
```

To check if the operation was performed successfully, other than looking at the output you can check the `HasWarnings` function of the `TJSONMarshal` class, with code like the following in which the `jtfiel d` local variable used by the for-each loop is of type `TTransi entFi el d`:

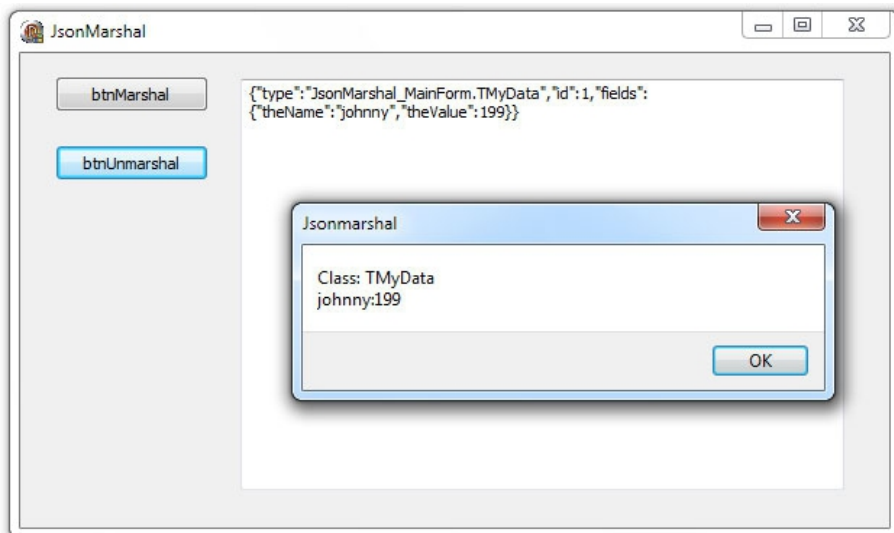
```
if j Marshal . HasWarni ngs then
begin
  Memo1. Li nes. Add(sLi neBreak + ' TJSONMarshal warni ngs: ');
  for j tfi el d i n j Marshal . Warni ngs do
    Memo1. Li nes. Add (j tfi el d. Fi el dName + ': ' + j tfi el d. TypeN ame);
```

This JSON representation can now be read into a `TJSONVal ue` and this can be re-converted into the original Delphi object, by creating an object of the qualified type indicated at the very beginning. The code is more or less the reverse of what we have just seen:

```
j Val ue := TJSONObj ect. ParseJSONVal ue(
  TEncodi ng. ASCI I . GetBytes (Memo1. Li nes. Text), 0);
j Unmarshal := TJSONUnMarshal . Create;
anObj ect := j Unmarshal . Unmarshal (j Val ue);
```

We can now use this object like a native Delphi object:

```
ShowMessage ('Cl ass: ' + anObj ect. Cl assName +
  sLi neBreak + anObj ect. ToStri ng);
```



The output of this operation is visible in the next page. Notice that I edited the text of the memo to recreate an object with slightly different data. At the end, remember to free all temporary objects (possibly in finally blocks):


```

anObject.Free;
jUnmarshal.Free;
jValue.Free;

```

Using JSON Converters and Reverters

The JSON marshaling support works great for basic types, but when data structures become complex, you might want to customize the actual information that is saved. This is possible thanks to the ability of installing custom converters and reverters which you can register for given classes and fields. The converters and reverters are implemented using anonymous methods⁹⁸.

Suppose, as an example taken from the same JsonMarshal demo, that you have a class like the following:

```

type
  TDataWithList = class
  private
    theName: String;
    theList: TStringList;
  public
    constructor Create (const aName: string); overload;
    constructor Create; overload;
    function ToString: string; override;
    destructor Destroy; override;
  end;

```

The constructors and destructors create and free the internal string list object. Notice that we need a parameter-less constructor as this is the one that will be invoked when the object is created through RTTI support. The constructor with the parameter, instead, initializes the string list with random numeric values.

Now if we write code similar to the previous portion of the example to create a JSON representation of the class, we'll get something like:

```

{
  "type": "JsonMarshal_MainForm.TDataWithList",
  "id": 1,
  "fields": {
    "theName": "john",
    "theList": {
      "type": "Classes.TStringList",
      "id": 2,
      "fields": {
        "FCount": 10,
        "FCapacity": 12,

```

98 For detail on anonymous methods see my “Delphi 2009 Handbook”.

```

        "FSorted": false,
        "FDuplicates": "dupl ignore",
        "FCaseSensitive": false,
        "FOwnsObject": false,
        "FDelimiter": "",
        "FLineBreak": "",
        "FQuoteChar": "",
        "FNameValueSeparator": "",
        "FStrictDelimiter": false,
        "FUpdateCount": 0
    }
}
}
}
}

```

Sorry for the long listing, not terribly interesting, but I wanted to underline that there are all sorts of various private fields in the `TStringList` object, including many internal ones that make little sense (like the `Capacity`), but there is a significant omission: the elements of the string list are missing!

Not only will recreating this object cause an exception... but you'll even be unable to get back the actual data. The solution, as anticipated, is to customize the JSON output by defining a custom converter. You can convert a data structure to any `TJSONValue` or map it to one that the marshaling layer can manage.

In this case I've decided to convert the `TStringList` object into an array of strings, which is how a `TListOfStrings` is defined. This is the complete code for the marshaling operation:

```

procedure TFormJson. btnMarshalConverterClick(Sender: TObject);
var
    theData: TDataWithList;
    jMarshal: TJSONMarshal;
    jValue: TJSONValue;
begin
    theData := TDataWithList.Create('john');
    try
        jMarshal := TJSONMarshal.Create(TJSONConverter.Create);
        try
            jMarshal.RegisterConverter(TDataWithList, 'theList',
                function (Data: TObject; Field: string): TListOfStrings
                var
                    l: Integer;
                    sList: TStringList;
                begin
                    sList := TDataWithList(Data).theList;
                    SetLength(Result, sList.Count);
                    for l := 0 to sList.Count - 1 do
                        Result[l] := sList[l];
                    end);
        finally
            jValue := jMarshal.Marshal(theData);
        end
    finally
        theData.Free;
    end

```

```

    try
      Memo1.Lines.Text := jValue.ToString;
    finally
      jValue.Free;
    end;
  finally
    jMarshal.Free;
  end;
finally
  theData.Free;
end;
end;

```

Now the JSON representation for our object becomes:

```

{
  "type": "JsonMarshal_MainForm.TDataWithList",
  "id": 1,
  "fields": {
    "theName": "john",
    "theList": ["588", "31", "656", "489", "693",
               "631", "742", "816", "166", "977"]
  }
}

```

The opposite operation is performed by registering a reverter, another anonymous method, which will receive the array of strings and populate the `TStringList` object. Again, this is the complete de-marshaling code

```

procedure TFormJson.btnUnmarshalReverterClick(Sender: TObject);
var
  jUnmarshal: TJSONUnMarshal;
  jValue: TJSONValue;
  anObject: TObject;
begin
  jValue := TJSONObject.ParseJSONValue(
    TEncoding.ASCII.GetBytes(Memo1.Lines.Text), 0);
  try
    jUnmarshal := TJSONUnMarshal.Create;
    try
      jUnmarshal.RegisterReverter(TDataWithList, 'theList',
        procedure (Data: TObject; Field: string;
          Args: TListofStrings)
        var
          I: Integer;
          sList: TStringList;
          begin
            sList := TDataWithList(Data).theList;
            for I := 0 to Length(Args) - 1 do
              sList.Add(Args[I]);
            end);
      anObject := jUnmarshal.Unmarshal(jValue);
    try
      ShowMessage('Class: ' + anObject.ClassName +
        sLineBreak + anObject.ToString);
    finally
      anObject.Free;
    end
  finally
    jUnmarshal.Free;
  end

```

```

    finally
        anObject.Free;
    end;
    finally
        jUnmarshal.Free;
    end;
    finally
        jValue.Free;
    end;
end;

```

JSON Values and Marshaling in DataSnap Server Methods

Now that we know how to create a JSON value for different types and convert this representation to and from Delphi objects, we can apply it to a DataSnap server, defining methods that receive or return JSON-based parameters and Delphi objects (using marshaling).

The Server and the Client Applications

The server class used by this example (a TCP/IP based DataSnap server targeting a `TPersistent` class and called `DSnapJson`) is the following:

```

type
{$METHODINFO ON}
    TDSnapJsonMethods = class(TPersistent)
    public
        function SimpleValue: TJSONValue;
        function GetList (nElem: Integer): TJSONArray;
        function GetData (const strName: string): TJSONValue;
    end;
{$METHODINFO OFF}

```

The `TJSONValue` returned by the `SimpleValue` method is a plain value, while the `TJSONValue` returned by the `GetData` method is a marshaled instance of the `TMyData` class. This is the class I used in the earlier section “Streaming Objects to JSON” and it is defined in a separate unit, as I’ll also need to compile the class in the client application, to be able to de-marshal the JSON data and rebuild the object. The third method returns a specific JSON data structure, an array or `TJSONArray`.

The client side application is a plain VCL program, which has an `SQLConnection` component hooked to the server. In this client project, I’ve created the client proxy for the server methods, which looks like the following:

```

type
  TDSnapJsonMethodsClient = class
  private
    FDBXConnection: TDBXConnection;
    FInstanceOwner: Boolean;
    FSimpleValueCommand: TDBXCommand;
    FGetListCommand: TDBXCommand;
    FGetDataCommand: TDBXCommand;
  public
    constructor Create(ADBXConnection: TDBXConnection); overload;
    constructor Create(ADBXConnection: TDBXConnection;
      AInstanceOwner: Boolean); overload;
    destructor Destroy; override;
    function SimpleValue: TJSONValue;
    function GetList(nElem: Integer): TJSONArray;
    function GetData(strName: string): TJSONValue;
  end;

```

Memory Management for JSON Values

In the proxy class there is an option to control the ownership of the JSON object instances (`FInstanceOwner`) being received from function calls, so that when you call a method and receive a `TJSONValue` result you don't have to remember to manually free it.

Likewise, when the server returns a JSON object it is DataSnap's responsibility to free the objects it has returned. That means that we don't have to worry about freeing `TJSONValue` or derived objects on the server as well.

What if you need to use these objects outside of the method that returned or received it? You can use the `Clone` method of `TJSONValue`, to create a new instance with the same value. For example on the server, if you need to return an object you still need to keep around, you can create and return a clone to that object, and the DataSnap infrastructure will take care of freeing the clone whilst you are responsible for the original object.

Passing JSON Values Manually

Now that I have covered the server side and client side infrastructure, let me focus on the method calls one by one. The first server method, `SimpleValue`, returns a fixed and hand-crafted data structure, of type `TJSONValue`:

```

function TDSnapJsonMethods.SimpleValue: TJSONValue;
begin
  Result := TJSONObject.Create (
    TJSONPair.Create ('name', 'Marco'));
  (Result as TJSONObject).AddPair(
    TJSONPair.Create ('email', 'marco@marcocantu.com'));
end;

```

The corresponding client calls (based on the proxy object) use the JSON Value directly (remember, no need to destroy it):

```
procedure TFormJsonClient.btnJsonValueClick(Sender: TObject);
var
  jValue: TJSONValue;
begin
  jValue := MethodsProxy.SimpleValue;
  Log(jValue.ToString);
end;
```

The result of this call is the JSON representation of the JSON data structure created on the server:

```
{ "name": "Marco", "email": "marco@marcocantu.com" }
```

Returning an Array of Elements

The second method returns a list of elements (in this case a list of strings) in a TJSONArray, that is created dynamically and returned:

```
function TDSnapJsonMethods.GetList(nElem: Integer): TJSONArray;
var
  l: Integer;
begin
  Result := TJSONArray.Create;
  for l := 1 to nElem do
    Result.Add('Item ' + IntToStr(l));
end;
```

The client passes the number of elements of the list, using a plain data structure (I could have used a TJSONNumber to pass the value, but better use the Integer type). This is a sample call, asking for an array with five elements:

```
procedure TFormJsonClient.btnArrayClick(Sender: TObject);
var
  jArray: TJSONArray;
begin
  jArray := MethodsProxy.GetList(5);
  Log(jArray.ToString);
end;
```

The output of this call is an array in its JSON representation:

```
[ "Item 1", "Item 2", "Item 3", "Item 4", "Item 5" ]
```

Passing a Marshaled Delphi Object

The last method is the most interesting one, as it lets you move a Delphi object from the server to the client. What we are moving, in fact, is the data of the object, not the code of its methods. These objects, in fact, must be compiled

(with the same structure) both in the client and the server, and the new Extended RTTI must see same the class layout when it maps the object data to JSON and the JSON to the object data.

Of course, this cannot be done directly, but requires the marshaling support I've already covered in the previous section. In the server side code of the `GetData` method notice that while the memory for the returned JSON value is managed, you have to remember to free any temporary object and the marshaling object:

```
function TDSnapJsonMethods.GetData(
  const strName: string): TJSONValue;
var
  myData: TMyData;
  jMarshal: TJSONMarshal;
begin
  myData := TMyData.Create(strName);
  try
    jMarshal := TJSONMarshal.Create(TJSONConverter.Create);
    try
      Result := jMarshal.Marshal(myData);
    finally
      jMarshal.Free;
    end;
  finally
    myData.Free;
  end;
end;
```

In case you want to pass complex data structures you'll have to register the proper converters, as covered in the previous section about JSON. In the corresponding client call we have to de-marshal the JSON representation to obtain the original Delphi object (provided the class is indeed compiled into the client program):

```
procedure TFormJsonClient.btnMarshalClick(Sender: TObject);
var
  jValue: TJSONValue;
  jUnmarshal: TJSONUnMarshal;
  myData: TMyData;
begin
  jValue := MethodsProxi.GetData('joe');
  jUnmarshal := TJSONUnMarshal.Create;
  try
    myData := jUnmarshal.Unmarshal(jValue) as TMyData;
    try
      Log(myData.ToString);
    finally
      myData.Free;
    end;
  finally
    jUnmarshal.Free;
  end;
end;
```

```

    j Unmarshal . Free;
  end;
end;

```

Again, the marshaling support objects and the actual Delphi object you re-create must be manually freed once you don't need them any more. The output shouldn't be surprising (beside the fact you'll get a different value for each object created on the server):

```

j oe: 444

```

Server Methods Callbacks

One of the problems with server methods is that they might take some time to complete, thus blocking the client application which has no way to figure out if there is any work in progress or the server is just stuck.

While covering DataSnap in my Delphi 2009 Handbook, I wrote an example showing the advantage of encapsulating the client call in a thread, so that even if the server takes time to respond and we don't know what is going on, at least the user interface of the client application remains responsive. In the opposite case, not using threads, the user interface freezes until we get a response (or a time-out error). You'll find the source code of this threaded example in the DsnapMethodsCallback example⁹⁹, which extends the program written for Delphi 2009.

What is new in DataSnap in Delphi 2010 is the ability to send information from the server back to the client while it is waiting for a response. A callback in DataSnap is not a mechanism to let the server call the client at will (also because this would make very little sense in case of a stateless HTTP communication layer), but it is limited only to the execution of the server method¹⁰⁰.

99 The example is somewhat complex, and has a client capable of starting a local instance of the server with different session lifetime configurations. Here I won't describe all of the features of the program in details but focus only on the specific issue, calling a slow server method.

100 In theory, one could write a very long or even infinite server method, which will let the server keep calling the client indefinitely. I'm not convinced this is a great idea in general, compared to letting the client poll the server at regular intervals, but there might be specific situations in which infinite server methods with a callback come in handy.

The Server Side Implementation of a Callback

Having clarified the scope, let us look at the actual implementation. On the server, the server class is declared as:

```
type
{$MethodInfo ON}
TSimpleServerClass = class(TPersistent)
public
    function Echo (const Text: string): string;
    function SlowPrime (MaxValue: Integer): Integer;
    function SlowPrimeCallback (MaxValue: Integer;
        aCallback: TDBXCallback): Integer;
end;
{$MethodInfo OFF}
```

As you can see there is the original `SlowPrime` function, which takes several seconds to execute, and the `SlowPrimeCallback` version with the *callback* parameter. It is this new one I am focusing on here.

The server method receives as extra parameter, an object implementing the `TDBXCallback` class and can use its `Execute` method to send information back to the client while the original function call is still taking place. This is the declaration of the virtual abstract method of the `TDBXCallback` class:

```
type
TDBXCallback = class abstract
public
    function Execute(const Arg: TJSONValue):
        TJSONValue; virtual; abstract;
```

As the server passes some data back to the client, it receives a return value. You can use it as you like, but the general idea is to use the return value of the `Execute` method of the callback to let the client application ask the server to “stop the execution”. That is, while the server is processing, it can tell the client it is progressing along and ask whether to continue or not.

This approach provides a nice way to let the user of the client application press a Cancel button in case the operation is too slow, notify the server and ask the server to stop the long operation. The standard implementation is to use the `TJSONFalse` value to ask the server to stop.

This is also the approach used by the `SlowPrimeCallback` function, which checks for a `TJSONFalse` return value:

```
function TSimpleServerClass.SlowPrimeCallback(
    MaxValue: Integer; aCallback: TDBXCallback): Integer;
var
    I: Integer;
```

```

    jsonResult: TJSONValue;
    isFalse: Boolean;
begin
    // counts the prime numbers below the given value
    Result := 0;
    for I := 1 to MaxValue do
    begin
        if IsPrime (I) then
            Inc (Result);
        if (I mod 100) = 0 then
        begin
            jsonResult := aCallback.Execute(TJSONNumber.Create(I));
            isFalse := jsonResult is TJSONFalse;
            jsonResult.Free;
            if isFalse then
                Break;
        end;
    end;
    aCallback.Free;
end;

```

Every 100 cycles in the loop (calling back the client for every single cycle would be overkill) the program calls the Execute method. Notice that both the callback object and the parameter returned by *executing* the callback must be freed manually, or you'll experience a memory leak.

The code has an extra intermediate variable, `isFalse`, as I cannot break out of the loop before freeing the `TJSONValue` object and I cannot check its value after it has been destroyed.

The Client Side Implementation of a Callback

This is all you have to do on the server. On the client side, on the other hand, you have to provide an actual implementation of the Execute method, providing an actual implementation for the virtual abstract class:

```

type
    TMyCallback = class(TDBXCallback)
    private
        fLabel: TLabel;
    public
        constructor Create (aLabel: TLabel);
        function Execute(const Arg: TJSONValue):
            TJSONValue; override;
    end;

```

The actual code is in the Execute method, which updates a label displaying the current status, lets the client application process Windows messages (thus avoiding freezing the user interface even without using a thread), and returning

the indication whether the server can keep going (TJSONTrue) or should stop (TJSONFalse):

```
function TmyCallBack.Execute(
  const Arg: TJSONValue): TJSONValue;
begin
  fLabel.Caption := Arg.ToString;
  Application.ProcessMessages;

  if fLabel.Tag = 0 then
    Result := TJSONTrue.Create
  else
    Result := TJSONFalse.Create;
end;
```

The main form use the label's Tag to communicate with the object implementing the callback. This is set to zero before starting the call (see the coming listing of the btnPrimesCallClick method) and set to one as the user clicks on the label itself.

With this class available, all you have to do on the client side, when you call the server method¹⁰¹, is to pass an instance of the callback implementation:

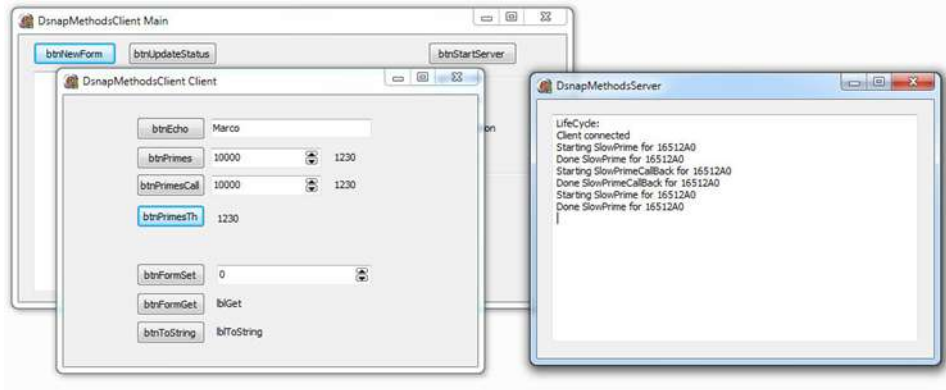
```
procedure TFormDsmcClient.btnPrimesCallClick(Sender: TObject);
begin
  if not Assigned (SimpleServer) then
    SimpleServer := TSimpleServerClassClient.Create (
      SQLConnection1.DBXConnection);
  lblPrimesCall.Tag := 0; // reset
  lblPrimesCall.Caption := IntToStr (
    SimpleServer.SlowPrimeCallback (SpinEdit3.Value,
      TmyCallBack.Create (lblPrimesCall)));
end;
```

Now we can run the program and test if it works. You can simply start the server, start the client, and press the btnNewForm button to open an actual client form with a connection.

From the client form you can call the slow server side method in a standard *blocking* way (the btnPrimes button), use the callback (the btnPrimesCall button), or within a thread (the btnPrimesTh button). In case of the call back, you can click on the label next to the button to stop server side processing. The server logs the request, so you can better understand what is going on.

101 In this application I'm using the DataSnap client proxy to call the server methods, as is much better than relying on data set parameters for complex calls.

The server form is visible here on the right, while on the left there is the Main client form along with the actual Client form of the client with the connection and the button to call server methods.



What's Next

Now that we have explored DataSnap in some detail, there is one area of this architecture that I mentioned and didn't delve into. This is the REST support offered by DataSnap and tied to its JSON support.

If REST architectures are generally quite simple to understand, using REST in Delphi opens up communication with many online services and the ability to provide the back-end to JavaScript browser-based applications. Rather than focusing exclusively on new features of Delphi 2010, in the last chapter of the book I'll open up the coverage to one of the fastest growing areas of today's technology, namely cloud and service-based architectures.

Chapter 8: REST Web Services

Over the last few years, there has been a large expansion of and a significant shift around web services. Promised as a new way for applications to interact and exchange data, web services have long been synonymous with the SOAP technology behind them, but have been quite slow to catch up with their promises. The recent expansion in web services, both as a way to interact with major web sites and with remote applications, is mostly due to a competing technology called Representational State Transfer, generally referred to as REST.

In this chapter I'll introduce REST to you, focusing on how it is supported in Delphi 2010 and opening up the coverage to the development of client REST applications and to building REST servers in Delphi with a direct, manual approach not based on DataSnap.

I'll also cover a few Delphi 2010 related topics, like the support for version 1.2 of the SOAP standard and improvements in the native XML processing support, and other topics tied to REST but not specifically to Delphi 2010, like XML processing techniques and JavaScript development. I won't delve into the JavaScript language by itself, but focus on the use of the jQuery library.

Why Web Services?

The rapidly emerging web services technology has the potential to change the way the Internet works for businesses. Browsing web pages to enter orders is fine for individuals (business-to-consumer applications) but not for companies (business-to-business applications). If you want to buy a few books, going to a book vendor's website and punching in your requests is probably fine. But if you run a bookstore and want to place hundreds of orders a day, this is a far from efficient approach, particularly if you have a program that helps you track your sales and determine reorders. Grabbing the output of this program and re-entering it into another application is ridiculous.

Web services are meant (or to be more precise were originally meant) to solve this issue: The program used to track sales can automatically create a request and send it to a web service, which can immediately return information about the order. The next step might be to ask for a tracking number for the shipment. At this point, your program can use another web service to track the shipment until it is at its destination, so you can tell your customers how long they have to wait. As the shipment arrives, your program can send a reminder via SMS or pager or Twitter to the people with pending orders, issue a payment with a bank web service, and... I could continue but I think I've given you the idea. Web services are meant for computer interoperability, much as the Web and e-mail let people interact.

The topic of web services is broad and involves many technologies and business-related standards. As usual, I'll focus on the underlying Delphi implementation and the technical side of web services, rather than discuss the larger picture and business implications. Delphi for Win32 offers some rather sophisticated support for web services, which originally came in the form of SOAP, and can now easily be extended by means of HTTP components and REST.

Web Service Technologies: SOAP vs. REST

The idea of a web service is rather abstract. When it comes to technologies, there are currently two main solutions that are attracting developers. One is the use of the SOAP standard (Simple Object Access Protocol, see the reference site

at <http://www.w3.org/TR/soap/>), another is the use of a REST (Representational State Transfer) approach, along with its variation XML-RPC (XML-Remote Procedure Call).

What is relevant to notice is that both solutions generally use HTTP as the transmission protocol (although they do provide alternatives) and use XML (or JSON) for moving the data back and forth. By using standard HTTP, a web server can handle the requests, and the related data packets can pass through firewalls.

In this chapter I won't provide many details on SOAP (with the exclusion of mentioning new features added to Delphi 2010), but focus extensively on REST. I'll start by providing some theoretical foundations, show a simple "hand-made" example of a server and a client, delve into the development of REST clients for popular REST web services and focus on the REST server side support available in Delphi 2010 as an extension of the DataSnap architecture.

XML and SOAP Updates

Before I start focusing on REST and *companion* technologies, let me cover a couple of relevant Delphi 2010 updates that relate to Web Services in general. The first is the improved support for XML processing, with updates in MS XML DOM mapping and in the version of the native OpenXML library that ships with the product. The second is the extension in SOAP support, specifically compatibility with version 1.2.

XML Processing in Delphi 2010

Delphi's overall support for XML processing has not changed in any radical way, with support for DOM¹⁰² manipulation taking place through the XMLDocument component. This component offers the standard DOM interface, a

102 DOM stands for Document Object Model and is a standard for navigating a document represented as a tree of nodes. The DOM interface is a standard, even if very low-level way, to access documents such as an XML document or the HTML data of a page inside a browser.

higher level XML DOM, and interface binding through the XML Mapper tool. This component can interface with multiple XML DOM engines, some of which have seen significant improvements, as covered below.

Beside these important changes in the XML DOM libraries that are most used in Delphi applications, there are also improvements in the XML Data Binding wizard, which now handles the *include* element of schema files.

Microsoft XML DOM Version 6

The msxml unit interfacing Microsoft XML DOM (and available in the source\Win32\rtl\win folder) now refers to version 6 of the XML engine (msxml6.dll). Differently from the past¹⁰³, the unit now includes almost all interfaces, including the VB version of the SAX interface.

The unit with all interfaces is called msxml and its source code can be found under the source\Win32\rtl\win folder and not the source\Win32\xml folder, which hosts the DOM mappers, including the msxmldom unit for MS XML DOM mapping from the XMLDocument component.

As this is the library used by the Delphi IDE, its improvements also relate to the XML-related tool set of Delphi.

The Alternative Document Object Model

The Pascal native XML engine, written and maintained by Dieter Kohler, has been upgraded to the much newer version 4.3 (it had remain at version 2.3.14 for many years). The new library has also a different name: from “Open XML” it is now “Alternative Document Object Model”. Along side this change, the vendor name in the XMLDocument component was updated to “ADOM XML v4”, so you'll have to update any application using it.

This also implies a change in the unit structure, potentially causing incompatibilities. In the past there was only one unit (xdom), now there is a dozen of them (the main one being AdomCore_4_3). If you have delved into this library, of course, you'll see some quite radical changes.

103 In the past you had to generate these interfaces by importing the type library, as I did many times in the past for working with the SAX interface (covered by an example in Mastering Delphi 2005). Note, by the way, that you have to use the VB version of the SAX interface, because the C++ version is broken at the type library level.

Updating My LargeXml Application

To better figure out the effect of the XML-related changes, I've decided to upgrade an XML-processing application I wrote for the Mastering Delphi book series, called LargeXml, to Delphi 2010. This program uses both MSXML and ADOM, and their SAX interfaces quite extensively.

In short, the application connects to a database, generates ClientDataSets with a lot of data, and saves them to XML files with different techniques (direct Xml Data content of the component, XML mapper, manual creation of the XML format). The second feature of the demo, which is the one I'll focus on here, is to ability reopen these XML documents either in a DOM or by parsing them and filling a second ClientDataSet. The original parsing style was SAX¹⁰⁴.

The first think I had to do was to change the reference to OpenXML to ADOM XML in one of the XmlDocument components:

```
object XmlDocument2: TXMLDocument
  DOMVendorDesc = 'ADOM XML v4'
end
```

Next I had to make quite a few changes in the uses statements, replacing:

- the xdom unit with the AdomCore_4_3 unit
- the oxmldom unit (for using the Open XML DOM with the XmlDocument component) with the new adomxmldom interfacing ADOM
- the MSXML2_TLB interface unit with the ready to use msxml unit.

The other problem was that (at least in my installation) most of the units related to XML support are not available in dcu format in the lib folder, so I had to include their source code folder into the project Search path.

Next I had to fix the streaming code to support Unicode, for example changing the TFileStream based code to use the TStreamWriter class introduced in Delphi 2009. This is an example of the new coding style:

```
procedure TForm1.btnSaveXmlPacketClick(Sender: TObject);
var
  sWriter: TStreamWriter;
begin
  sWriter := TStreamWriter.Create ('data.xml', False {replace});
  try
    sWriter.Write (ClientDataSet1.XMLData);
```

¹⁰⁴ SAX, or Simple API for XML, is an event driven parsing technology. For each element of the XML file being parsed, the SAX engine will trigger an event (technically call a virtual method) which is up to to program to process or discard.

```

    finally
        sWriter.Free;
    end;
end;

```

The code for MS SAX support worked with basically no changes, as my original code used the VBSAX interfaces obtained by importing the type library (probably of MS XML version 4 at the time). All I had to do was change the name of the `Set_documentLocator` method to `_Set_documentLocator`, adding the initial underscore.

For the code in the ADOM XML I didn't find any equivalent to the original SAX-like support, so I basically had to rewrite that using the new signals-based coding style. The core of the parsing code is the `inProcessSignal` method of the `TXmlStandardHandler` derived class, which provides the custom implementation for the parsing code.

In this code the program handles three events (or signals). As it hits the start of an XML node (`TXmlStartElementSignal`), the program adds the node to a local stack (a string list) and when this is a new employee record it calls the `Insert` method of the target `ClientDataSet` component. The program keeps adding any textual elements (`TXmlPCDATASignal`) to the current text, which might be divided among multiple nodes. When it hits the end of an XML node (`TXmlEndElementSignal`), if it was reading an XML node at level 3 of nesting the program adds the text that was read to the corresponding field of the `ClientDataSet`, while if this is at the end of the record, it posts the data:

```

procedure TDataSaxHandler.ProcessSignal (const Signal : TXmlSignal);
var
    tagname: string;
begin
    if Signal is TXmlStartElementSignal then
        begin
            stack.Add (TXmlStartElementSignal (Signal).TagName);
            if TXmlStartElementSignal (Signal).TagName = 'employeeData' then
                Form1.clientdataset2.Insert;
            strCurrent := '';
        end
    else if Signal is TXmlEndElementSignal then
        begin
            if TXmlEndElementSignal (Signal).TagName = 'employeeData' then
                Form1.clientdataset2.Post;
            if stack.Count > 2 then
                begin
                    Form1.ClientDataSet2.Edit;
                    Form1.ClientDataSet2.FieldByName (
                        TXmlEndElementSignal (Signal).TagName).
                        AsString := strCurrent;
                end
            end
        end
    end

```

```

        end;
        stack.Delete (stack.Count - 1);
    end

    else if Signal is TXmlPCDATA_Signal then
    begin
        strCurrent := strCurrent + RemoveWhites(
            TXmlPCDATA_Signal (Signal).Data);
    end;
end;

```

Also the code to invoke this signal-based processing is slightly different from the original code used to invoke the SAX engine. Its essence is:

```

reader := TXmlStandardDocReader.Create (nil);
reader.NextHandler := TDataSaxHandler.Create (nil);

SourceStream := TFileStream.Create(Filename, fmOpenRead);
SysId := FilenameToUriWideStr(Filename, []);
inputSource := TXmlInputSource.Create(SourceStream,
    '', SysId, 4096, '', False, 0, 0, 0, 0, 1);
reader.Parse (inputSource);

```

Now if you've never used a SAX engine this code and description might seem a little awkward, but I felt it relevant to mention the change and show some demo code to everyone who used this XML library (maybe following my very own advice!).

SOAP 1.2 Support

Even if I think REST interfaces are on the rise and SOAP its in decline, there are significant environments (from governmental bodies to large companies) who mandate the use of SOAP as a way to communicate with them. That's why good quality SOAP support remains an important Delphi feature.

In the new version, Delphi adds support for the development of SOAP version 1.2 clients, which conform to the standard. You can now import newer WSDL files and generate the Object Pascal interfaces for newer services. I have done very limited testing of this aspect.

Note that SOAP support is now formally limited to the development of clients: building SOAP servers in Delphi is still possible but the technology has been deprecated. Finally, there is another minor but potentially interesting option, which is support for HTTPS requests authenticated using an X509 certificate. This is available using a new `InvokeOption` of the `HTTPRIO` component, `soPickFirstClientCertificate`.

What is REST?

Even if the general idea of REST has been around for some time, the introduction of this formal name and the theory behind it are fairly recent. What is relevant to mention up front is that there isn't a formal REST standard.

The term REST, an acronym for Representational State Transfer, was originally coined by Roy Fielding in his Ph.D. dissertation in year 2000, and spread very rapidly as a synonym for accessing data over the web using HTTP and URLs, rather than relying on the SOAP standard.

The term REST was originally used to describe an architectural style which described the relationship of a web browser with a server. The idea is that when you access a web resource (either using a browser or a specific client application) the server will send you a representation of the resource (an HTML page, an image, some raw data...). The client receiving the representation is set in a given state. As the client accesses further information or pages (maybe using a link) its state will change, transferring from the previous one. In Roy Fielding's words:

"Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

REST Architecture's Key Points

So, if REST is an architecture (or even better, an architectural style) it is clearly not a standard, although it uses several existing standards like HTTP, URL, plus many format types for the actual data.

While SOAP relies on HTTP and XML but builds on those, REST architectures use HTTP and XML (or other formats) exactly as they are:

- REST uses URLs to identify a resource on a server (while SOAP uses a single URL for many requests, detailed in the SOAP envelope). Notice the idea is to use the URL to identify a resource not an operation on the resource.

- REST uses HTTP methods to indicate which operation to perform (retrieve or HTTP GET, create or HTTP PUT, update or HTTP POST, and delete or HTTP DELETE).
- REST uses HTTP parameters (both as query parameters and POST parameters) to provide further information to the server.
- REST relies on HTTP for authentication, encryption, security (with HTTPS).
- REST returns data as plain documents, using multiple mime formats (XML, JSON, images, and many others).

There are quite a few architectural elements that are worth considering in this kind of scenario. REST demands for systems to be:

- Client/server in nature (nothing directly to do with database RDBMS here).
- Inherently stateless.
- Cache-friendly (the same URL should return the same data if called twice in sequence, unless the server side data changed), permitting proxy and cache servers to be inserted between the client and the server. A corollary is that all GET operations should have no side effect.

There is certainly much more to the theory of REST than this short section covered, but I hope this got you started with the theory. The practical examples coming next along with Delphi code should clarify the main concepts.

The REST Architecture and Delphi

Having said that there is no REST standard and that you need specific tools for REST development, there are existing standards that REST relies upon and that are worth introducing (an in-depth description of each could take an entire book). The specific focus here is Delphi support for these technologies.

HTTP, Client and Server

The HyperText Transfer Protocol is the standard at the heart of the World Wide Web, and needs no introduction. Granted, HTTP can be used by Web Browsers, but also by any other application.

In Delphi applications the simplest way to write a client application that uses HTTP is to rely on the Indy HTTP client component, or `IdHttp`. If you call the `Get` method of this component, providing a URL as parameter, you can retrieve

the content of any Web page and many REST servers¹⁰⁵. At times, you might need to set other properties, providing authentication information or attach a second component for SSL support (as we'll see in some examples). The component supports all HTTP methods, in addition to Get.

On the server side you can use multiple architectures for creating a web server or web server extension in Delphi. For a stand-alone web server you can use the `IdHttpServer` component, while for creating web server extensions (CGI applications, ISAPI, or Apache modules) you can use the `WebBroker` framework. Another new option is given by the HTTP support within `DataSnap` in Delphi 2010. I partially covered that in the last chapter and will focus on it in this one.

XML

Extensible Markup Language is a commonly used format for data, although many REST servers use alternative data structures like JSON (JavaScript Object Notation) and at times even plain comma-delimited text files. Again, XML is widely used and I don't want to cover it in detail here.

In Delphi, you can process XML documents using the `XMLDocument` component, as covered in the earlier section on Delphi XML updates. The `XMLDocument` component is a wrapper around one of the available XML DOM engines (the default one being Microsoft XML DOM). Once a document is loaded you can navigate its node structure or query the document using XPath (which is often the style I prefer).

XPath

XPath is a query language that lets you identify and process the nodes of an XML document. The notation used by XPath resembles file system paths (`/root/node1/node2`) with square brackets added to express conditions on node attributes or subnodes (`root/node1[@val=5]`) or even complex expressions. The result of an XPath statement can itself be an expression, like the number of nodes matching a rule or the total value of a set of nodes.

¹⁰⁵ Notice that to be on the safe side you should generally make `IdHttp` requests inside a thread, as `Indy` uses blocking threads: the user interface of your program will be stuck until the requests are returned (which take a long time in case of a slow web server or a large data transfer). I won't generally use threads in the demos in this chapter, that is only for the sake of simplicity. I strongly recommend the use of threads in live applications.

In Delphi you can execute an XPath request by applying it to the DOM hosting the document, if the specific DOM supports it. We'll see an example of XPath in the first REST client demo.

REST Clients Written in Delphi

There are countless examples of REST servers that you can find on the Web. Even if the number of web services that uses REST on the Internet is high, most actual web services require some developer token (as covered in some of the coming demos), while only a handful offer totally free and open access. For a much longer list of Delphi REST clients that I have written, you can refer to the specific section of one of my web sites:

■ <http://ajax.marcocantu.com/delphi/rest>

You can also find more coverage of REST clients written in Delphi in the paper on REST that I wrote for Embarcadero Technologies, which is based on the material of this chapter:

■ http://www.embarcadero-info.com/in_action/radstudio/rest.html

In this chapter I'll cover only a few selected examples: a first demo of accessing RSS feeds (which uses XML and XPath), two mapping demos (based on different return types), and a translation example which used JSON.

A REST Client for RSS Feeds

The most widespread format for distributing information as XML is the use of the RSS and ATOM feeds, mostly attached to blog and news sites, but equally usable for any data source.

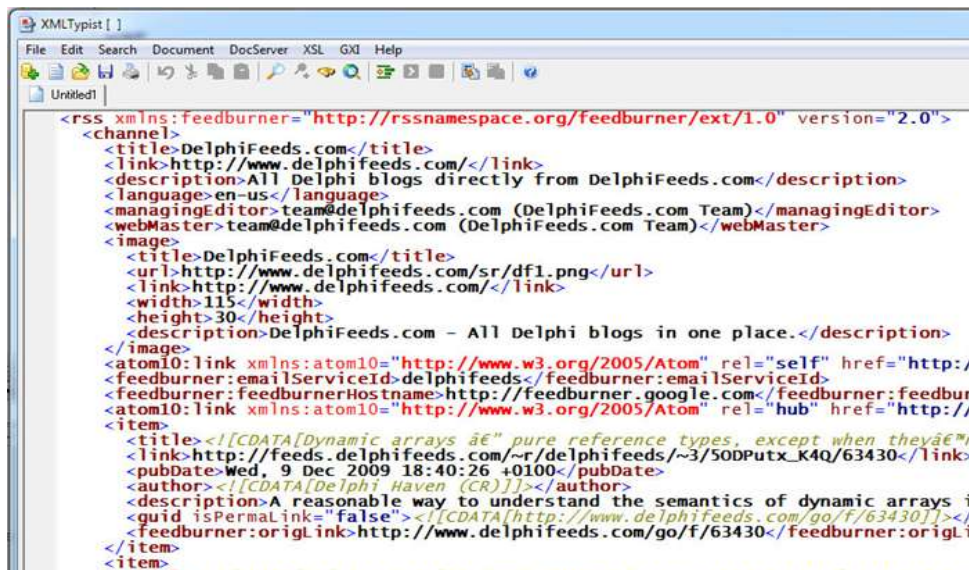
The interesting point about feeds is they provide the same information to client applications that a user will generally access using a web browser. Feed information is processed by these client applications, and at times even combined in a summary of similar feeds, as happens on the Delphi Feeds site:

■ <http://www.delphifeeds.com>

That's why, as a first example of a client application using REST, I wrote a very simple RSS client¹⁰⁶ looking into blogs at this site. Every time you access dynamic XML data using an URL and you can change the URL to access different data, you are using the REST approach. The RssClient program uses an IdHttp component and an XmlDocument component. The first component is used to grab the data from the Web and load it in the second component:

```
var
    strXml : string;
begin
    strXml := IdHTTP1.Get
        ('http://feeds.delphi.foxs.com/delphi/feeds');
    XMLDocument1.LoadFromXML(strXml);
```

The data extracted would look like the following (which I've somewhat simplified for readability), when displayed in an XML editor:



Processing the RSS Data with XPath

To extract the relevant information from this XML document the RssClient program uses XPath expressions. For example it reads the title of the first blog post (item) of the list uses the expression `/rss/channel/item[1]/title`.

106 A video of the development of this Delphi client application step-by-step and its final result is available on YouTube at <http://www.youtube.com/watch?v=b4cmBrqVRIA> and in my blog at http://blog.marcocantu.com/blog/rest_delphi_client_videos.html.

This is done in a cycle along with the extraction of some other information, formatted and displayed in a list box. Using XPath requires the use of a custom interface of the Microsoft engine: hence the cast to `IDOMNodeSelect`.

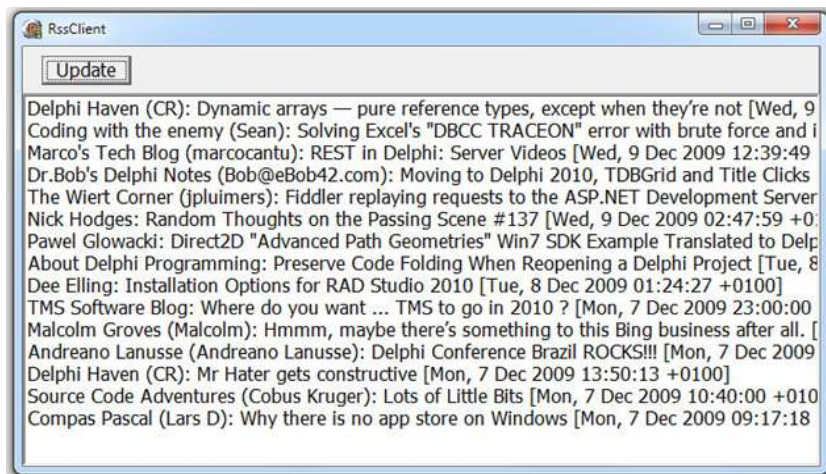
Once it has the nodes it is interested in, the program looks for any child text node, using a `getChildNodes` helper function I wrote for this purpose, and adds the data to a list box. This is the complete code of the method executed when the Update button of the program is pressed:

```

procedure TRssForm.btnUpdateClick(Sender: TObject);
var
    strXml, title, author, pubDate: string;
    I: Integer;
    IDomSel: IDOMNodeSelect;
    Node: IDOMNode;
begin
    strXml := IdHTTP1.Get
        ('http://feeds.delphi.com/delphi-feeds');
    XMLDocument1.LoadFromXML(strXml);
    XMLDocument1.Active := True;
    IDomSel := (XMLDocument1.DocumentElement.DOMNode
        as IDOMNodeSelect);
    for I := 1 to 15 do
        begin
            Node := IDomSel.selectNode(
                '/rss/channel/item[' + IntToStr(I) + ']/title');
            title := getChildNodes(Node);
            ...
            ListBox1.Items.Add(author + ': ' + title +
                ' [' + pubDate + ']');
        end;
    end;

```

The effect of running this program is visible here:



Of Maps and Locations

Access to location and map information can be very useful in multiple circumstances, as many applications have to do with addresses. In the recent years, more and more mapping data has been made available on the web by many large sites, including Google, Yahoo, and Microsoft.

Google Geocoding Service

The first service of this category I'm going to use is Google's Geocoding service, which lets you submit an address and retrieve its latitude and longitude:

```
http://maps.google.com/maps/geo?
q=[address]&output=[format] &key=[key]
```

You can also type a similar URL in your browser for testing purposes, as you can see here showing New York coordinates in a browser (in XML format):



The GeoLocation example¹⁰⁷ I've built uses the addresses of the companies of the classic Customer. cds sample database that comes with Delphi. As with many similar services, this is free for limited usage (the program has extra calls to the `sleep` procedure to slow it down and avoid hitting the maximum rate per minute), but requires a registration for the specific service on:

```
http://code.google.com
```

¹⁰⁷ A video with the output of both this and the next mapping example is available on YouTube at http://www.youtube.com/watch?v=C_saMKfP2wg and (again) also in my blog at http://blog.marcocantu.com/blog/rest_delphi_client_videos.html

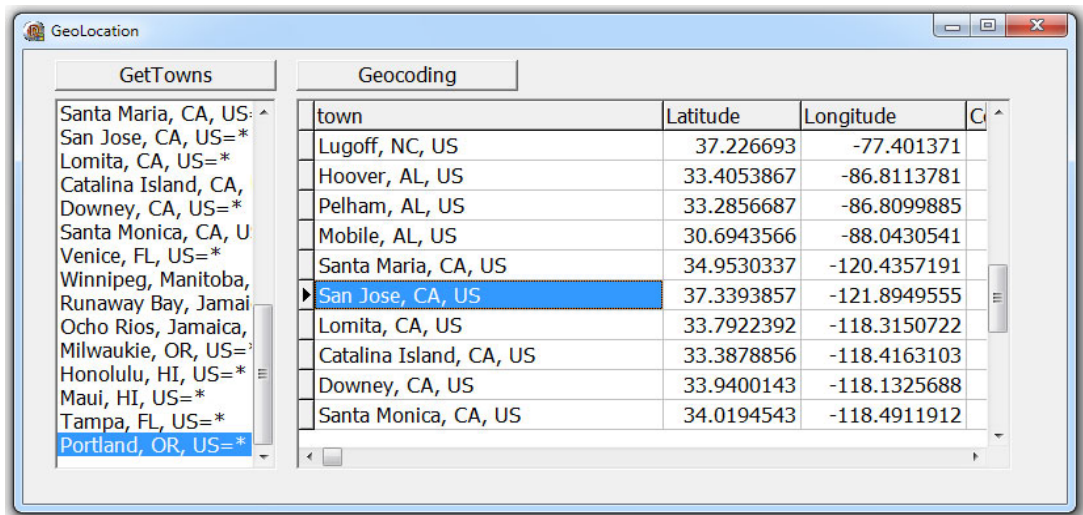
The demo program requires your *devkey* to be added to a *GeoLocation.ini* file which must reside in the user's document folder and has the simple structure:

```
[googlemap]
devkey=
```

Resolving Customer Addresses

The program works in two steps. First, it looks for unique names of cities / state / country, by scanning the *ClientDataSet* component and filling a string list. This code is unrelated to REST, so I've omitted it from the book¹⁰⁸.

The second step is to look up each city on the Google Geocoding service, filling an in-memory *ClientDataSet* with the resulting information:



This time rather than asking for the XML version of the data, I resorted to a simpler CSV format, which the program parses using a *TStringList* object. Here is the actual Geocoding code:

```
procedure TFormMap.btnGeocodingClick(Sender: TObject);
var
  I: Integer;
  strResponse, str1, str2: string;
  sList: TStringList;
begin
  cdsTown.Active := False;
  cdsTown.CreateDataSet;
  cdsTown.Active := True;
```

¹⁰⁸ Don't worry, it is in the book source code.

```

sList := TStringList.Create;
for I := 0 to sListCity.Count - 1 do
begin
  ListBox1.ItemIndex := I;
  if Length (sListCity.Names[I]) > 2 then
  begin
    strResponse := IdHTTP1.Get( TIDUri.UrlEncode(
      'http://maps.google.com/maps/geo?q=' +
      (sListCity.Names[I]) + '&output=csv&key=' +
      googleMapKey));
    sList.LineBreak := ',';
    sList.Text := strResponse;
    str1 := sList[2];
    str2 := sList[3];
    cdsTown.AppendRecord([sListCity.Names[I],
      StrToFloat (str1), StrToFloat (str2),
      Length (sListCity.ValueFromIndex[I])]);
    Sleep (150);
    Application.ProcessMessages;
  end;
end;
sList.Free;
end;

```

Yahoo Maps

As a further step, we can try to access to the actual map corresponding to an address. If Google Maps provide countless features, they are meant to be hosted on web sites not on client applications¹⁰⁹.

The new example, called YahooMaps uses Yahoo Map API to get an actual map and show it in an Image control. Information about this REST API and the link to obtain a free Yahoo Application ID are available at:

http://developer.yahoo.com/maps/

Again, to run the programs you'll have to obtain this ID and store in a specific INI file in the "user documents" folder called YahooMaps.ini. The map is retrieved in two steps: a first HTTP call passes the address and receives the URL of the map image, which is retrieved using a second HTTP call. Again, you could simulate the two steps in a web browser, for debugging purposes.

While the program uses the same database and intermediate StringList of the previous example, it also has a button that it uses to display the map or a hard-coded city (*San Jose, California*), using the following method:

¹⁰⁹ Although I do have an example of hosting a Google Map in a client program, its architecture and code are quite complex and the example won't fit in this chapter.

```

const
  BaseUrl = 'http://api.local.yahoo.com/MapsService/V1/';
procedure TFormMap.Button1Click(Sender: TObject);
var
  strResult: string;
  memStr: TFileStream;
begin
  strResult := IdHTTP1.Get(BaseUrl + 'mapImage?' +
    'appid=' + yahooAppid +
    '&city=SanJose,California');
  XmlDocument1.Active := False;
  XmlDocument1.XML.Text := strResult;
  XmlDocument1.Active := True;
  strResult := XmlDocument1.DocumentElement.NodeValue;
  XmlDocument1.Active := False;
  // now let's get the referred image
  memStr := TFileStream.Create('temp.png', fmCreate);
  IdHTTP1.Get(strResult, memStr);
  memStr.Free;
  // load the image
  Image1.Picture.LoadFromFile('temp.png');
end;

```

The first HTTP Get request provides the actual query and returns an XML document with the URL of the image of the actual map, which looks like:

```

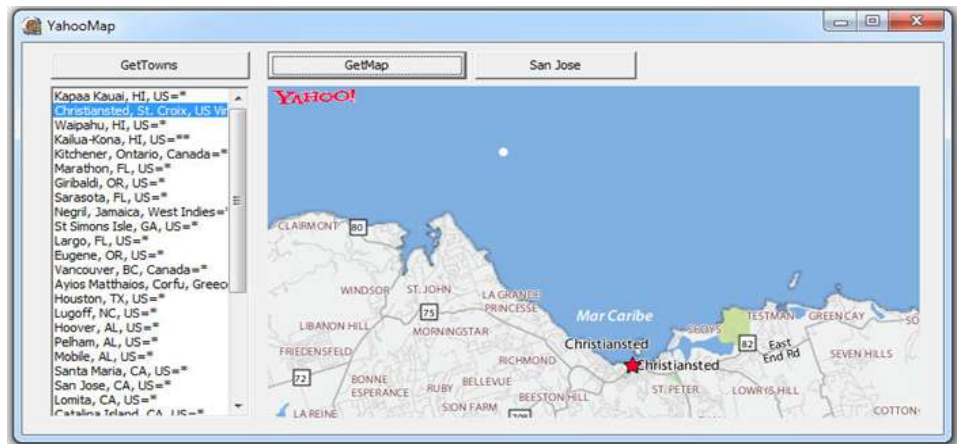
<Result>
  http://gws.maps.yahoo.com/mapImage?MAPDATA=[...]&mvmt=m
  &cltype=onnetwork&.intl=us&appid=[...]
  &oper=&_proxy=ydn.xml
</Result>

```

That's why the program can extract the value of the only node with the code:

```
XmlDocument1.DocumentElement.NodeValue
```

Finally, the image is saved to a file and loaded into an Image control:



Beside the map of this specific city, the program can also fetch those of the `Customer.cds` database of the previous example.

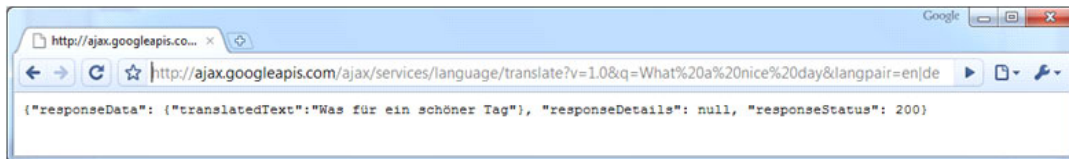
Google Translate API

Another simple and interesting example of a REST API provided by Google is their translation service, called Google Translate REST API. The documentation is at:

`http://code.google.com/apis/ajaxlanguage/documentation/`

In this case there is no need for a signup key (and an INI file), but only provide a referrer site (although everything seems to work even without that information). You can ask for a translation in your Web Browser by entering an URL like:

`http://ajax.googleapis.com/ajax/services/language/translate?v=1.0&q=What%20a%20nice%20day&langpair=en|de`



The output of this call is visible above (I have also listed the JSON result for readability):

```
{
  "responseData":
  {
    "translatedText": "Was für ein schöner Tag"
  },
  "responseDetails": null,
  "responseStatus": 200
}
```

This example takes one step further compared to previous demos. Rather than making the HTTP request, it uses a specific custom VCL component, invoked with a class method (so you don't need to place the component on a form, even if you could). This support component makes the API very easy to use, and encapsulates the HTTP call completely.

A Translation Component

The class of the component stores information about the request and encapsulates an internal HTTP client component, as you can see in its declaration:

```
type
  TBabel GoogleRest = class (TComponent)
  protected
    Http1: TIdHttp;
    FFromLang: string;
    FToLang: string;
    FActiveInForm: Boolean;
    procedure SetFromLang(const Value: string);
    procedure SetToLang(const Value: string);
  public
    function DoTranslate (strIn: string): string; virtual;
    constructor Create(AOwner: TComponent); override;
    class function Translate (
      strIn, langIn, langOut: string): string;
  published
    property FromLang: string read FFromLang write SetFromLang;
    property ToLang: string read FToLang write SetToLang;
end;
```

The actual processing (the REST call) is performed in the DoTranslate function, which uses the input and output languages set for the class:

```
function TBabel GoogleRest.DoTranslate(strIn: string): string;
var
  strUrl, strResult: string;
  nPosA, nPosB: Integer;
begin
  strUrl := Format (
    'http://ajax.googleapis.com/ajax/services/language/' +
    'translate?v=1.0&q=%s&langpair=%s',
    [TIdUri.ParamsEncode (strIn),
      FFromLang + '%7C' + FToLang]); // format hates %7
  strResult := Http1.Get(strUrl);
  Result := ResultFromJSONDirect (strResult);
end;
```

The result of the request to the given URL is in JSON format (as this is considered as a JavaScript API by Google). As I originally wrote this code for a past version of Delphi, the program parsed the JSON string to extract the actual result using direct string manipulation:

```
function TBabel GoogleRest.ResultFromJSONDirect(
  const strJson: string): string;
var
  nPosA, nPosB: Integer;
begin
  nPosA := Pos ('"translatedText":', strJson);
  if nPosA = 0 then
```



```

begin
  nPosA := Pos ('"responseDetails":', strJson);
  nPosA := nPosA + Length ('"responseDetails":');
end
else
  nPosA := nPosA + Length ('"translatedText":');

  nPosA := PosEx ('"', strJson, nPosA) + 1; // opening "
  nPosB := PosEx ('"', strJson, nPosA) - 1; // end "
  Result := Copy (strJson, nPosA, nPosB - nPosA + 1);
end;

```

As I moved the code to Delphi 2010, I tried using the using the new Delphi JSON support to accomplish the same, in a cleaner way. This is implemented in the alternative `ResultFromJSON` function, which actually highlights problems of the native JSON parsing. First, you have to remove white spaces in the JSON representation (but only outside of the quoted strings), or the parser will fail¹¹⁰. The *string cleaning up* operation is performed by a `RemoveWhites` function:

```

function RemoveWhites(const str1: string): string;
var
  ch: char;
  inQuotes: Boolean;
begin
  Result := '';
  inQuotes := False;
  for ch in str1 do
    begin
      if ch = '"' then
        inQuotes := not inQuotes;
      if inQuotes or (ch <> ' ') then
        Result := Result + ch;
    end;
  end;
end;

```

After this step the program gets the value of the first pair (*responseData*) of the JSON object, which is in turn an object, and read the string value of the first pair (*translatedText*) of this sub-object:

```

function TBabel GoogleRest.ResultFromJSON(
  const strJson: string): string;
var
  strTemp: string;
  jObject, jResponseData: TJSONObject;
begin
  // parse JSON using Delphi 2010 support
  strTemp := RemoveWhites (strJson);

```

110 The inability of parsing JSON with spaces is a bug and I have reported it on Quality Central with number 80262. The JSON RFC, in fact, specifically accounts for any white space within the JSON symbols. Seems this is going to be solved as part of a hotfix, which is still not available at the time I'm writing.

```

jObject := TJSONObject.ParseJSONValue(
  TEncoding.ASCII.GetBytes(strTemp), 0) as TJSONObject;
try
  if not Assigned(jObject) then
    Exit('Error parsing ' + strTemp);

    // read value of first pair of object, as subobject
  jResponseData := jObject.Get(0).JsonValue as TJSONObject;

    // get value of only element of responseData
  Result := jResponseData.Get(0).JsonValue.Value;
finally
  jObject.Free;
end;
end;

```

At the time writing, this code fails¹¹¹ for any Unicode string, which are a large majority of those managed in the demo program, considering the number of non-Latin languages Google Translation supports.

To perform the translation, the actual `DoTranslate` method can be invoked directly after creating an instance of the object and setting its properties, it can be called using the `Translate` class method. This function creates a temporary object, sets its properties, and calls the `DoTranslate` function on it:

```

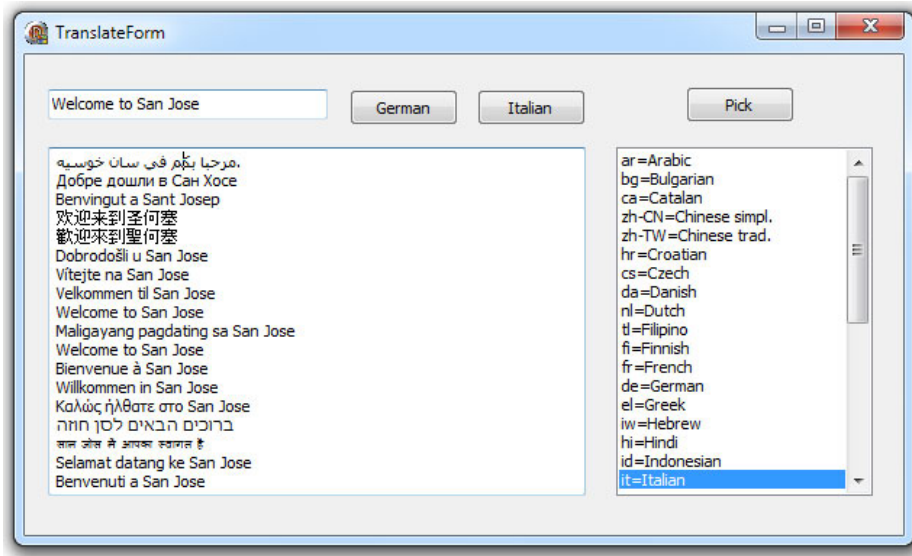
class function TBabelGoogleRest.Translate(const
  strIn, langIn, langOut: string): string;
var
  bg: TBabelGoogleRest;
begin
  bg := self.Create(nil);
  try
    bg.FromLang := langIn;
    bg.ToLang := langOut;
    Result := bg.DoTranslate(strIn);
  finally
    bg.Free;
  end;
end;

```

The main form of the demo program has a list box filled with all supported languages. The demo translates from English, but you can set it up in the opposite direction as well. In theory, any two language token combination works, in practice not always. Once you ask for a translation, the result is added to a log.

111 The `ParseJSONValue` method expects an array of bytes (representing characters) as parameter and will return an empty object in case of a Unicode string in input (whatever its content). If you convert the input to ANSI, any high character will not convert. If you convert it to UTF8, it will treat individual bytes as characters, which will not work, either. Again, it looks like this is going to be solved by a Delphi 2010 hotfix.

Here I've applied the call to the first group of languages (in alphabetic order), using direct JSON result string processing and not Delphi's JSON parser:



Building a REST Server

Now that we have spent a considerable amount of time looking at a few REST client applications written in Delphi, facing different permission requests and using different data type formats, we are ready to start delving into the second part of this chapter, focused on writing REST servers in Delphi 2010.

Just as building a REST client is more straightforward and requires less support from developer tools than building a SOAP client, the same can be said for the server. It is true that a SOAP server is a web server with an extra component for mapping requests to classes, but a REST server can be a plain web server extension with little additional code.

In this section I'll introduce the development of REST servers using a simple application built on top of the plain Web Broker architecture, something you could have done since Delphi 4. In the remaining part of the chapter, I'll focus on the development of REST servers based on the specific DataSnap extensions available in Delphi 2010.

To build the Rest1 project I created a standard web server using the Web App Debugger technology, already covered and used in the last chapter for some of the HTTP DataSnap applications. The web program has four actions, a sample *Echo* operation similar to the predefined DataSnap ones, and three database oriented requests. Each of the actions of the web modules has a corresponding OnAction event handler:

```

Actions = <
  item
    Default = True
    Name = 'actionEcho'
    PathInfo = '/Echo'
    OnAction = actionEchoAction
  end
  item
    Name = 'actionCustomers'
    PathInfo = '/Customers'
    OnAction = actionCustomersAction
  end
  item
    Name = 'actionCustData'
    PathInfo = '/CustData'
    OnAction = actionCustDataAction
  end
  item
    Name = 'actionCustomer'
    PathInfo = '/Customer/**'
    OnAction = actionCustomerAction
end>

```

An Echo Action

The actionEcho operation is a sample action mimicking the sample operation of DataSnap servers (including the REST ones, as we'll see). This operation has a parameter, passed with the *in* parameter name, and the response is sent back using a simple XML fragment and the corresponding HTTP response type:

```

procedure TWebModule3.actionEchoAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse;
  var Handled: Boolean);
var
  strInput: string;
begin
  strInput := Request.QueryFields.Values['in'];
  if strInput = '' then
    strInput := 'Nothing to echo';
  strInput := strInput + '...' + Copy(
    strInput, Length(strInput) - 4, 5);
  Response.Content := '<resul t>' + strInput + '</resul t>';
end;

```

```
Response.ContentType := 'text/xml';
end;
```

How can you call this operation? From a web browser, after we run the program and start the Web App Debugger, we can use the following URL:

```
http://localhost:8081/Rest1.rest1?in=hello%20world
```

The result will look like:

```
<result>hello world...world</result>
```

Nothing extraordinary but relatively simple. We can put the same URL in a Delphi client application (the Rest1Client example), passing the text of an edit box as parameter and encoding it:

```
const
  BaseUrl = 'http://localhost:8081/Rest1.rest1';

procedure TForm3.btnEchoClick(Sender: TObject);
var
  strInput, strResult: string;
begin
  strInput := TIdURI.ParamsEncode (edEcho.Text);
  strResult := IdHTTP1.Get(BaseUrl + '?in=' + strInput);
  lblEcho.Caption := DataFromTopTag (strResult);
end;
```

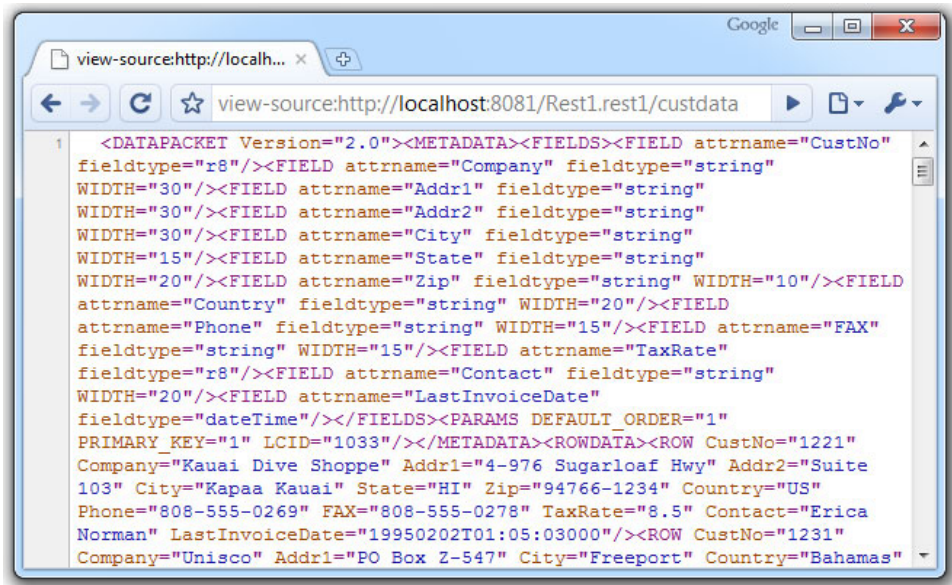
The DataFromTopTag support function (not listed here) removes the top and any XML tags from the response.

Returning the XML Data of a ClientDataSet

The second action, `actionCustData`, is the simplest to implement, on both the server and client side. All it does is return the entire data content of a ClientDataSet component, using its XML representation. The server simply returns that XML:

```
procedure TWebModule3.actionCustDataAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := cdsCustomers.XMLData;
  Response.ContentType := 'text/xml';
end;
```

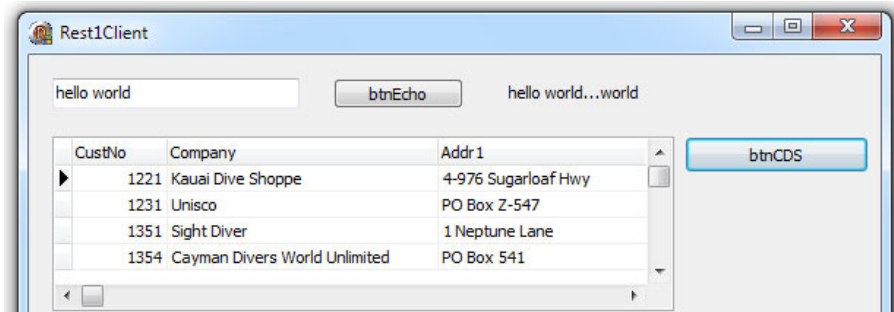
Using the corresponding URL you can see the entire XML data of the ClientDataSet in a browser, like in the following page:



A Delphi client application can easily access that data:

```
procedure TForm3. btnCDSCl i ck(Sender: TObj ect);
begin
  ClientDataSet1. Cl ose;
  ClientDataSet1. XMLData := IdHTTP1. Get(BaseUrl + '/custdata');
  ClientDataSet1. Open;
end;
```

In this client application the ClientDataSet is connected with a DBGrid, so you can immediately see the server side database in the client program, passed through the REST call. You can see the result of the first two operations in the top part of the form of the Rest1Client demo:



Implementing a client for this data access call in JavaScript or in a non-Delphi client might not be trivial, because the XML of the ClientDataSet component is

quite specific. Also, returning the entire data structure for a large data set might be excessive in terms of bandwidth. A more REST-oriented alternative might be to return a list of available records and then individual records with the data. This implies more processing both on the server and the client, but it is certainly worth exploring.

Returning a List of Customers

The server action returning a list is called with the *customers* URL. In its server code, the application scans the data set and returns an XML structure with the company name and the customer ID for each customer record. With the URL:

http: //l ocal host: 8081/Rest1. rest1/customers

You'll see an XML result like:

```
<customers>
  <customer>
    <i d>1221</i d>
    <Company>Kauai Di ve Shoppe</Company>
  </customer>
  <customer>
    <i d>1231</i d>
    <Company>Uni sco</Company>
  </customer>
  ...
```

This XML code is generated using as a helper the *TTrivialXmlWriter* class as a helper that I covered in Chapter 3, in the section “The Trivial XML Writer Class” under the heading “XML Streaming”. This is a class that let's you write to an XML stream or string, using the *TTextWriter* interface introduced in Delphi 2009. It keeps track of the XML nodes you open so it can close them in the reverse order, without specifying which tag you are closing.

In this specific case, the program uses a *TStringWriter*, as we are returning a string. The server code scans the data set component from the first to the last record and outputs the customer ID and company name:

```
procedure TWebModule3. acti onCustomersActi on(
  Sender: TObje ct; Request: TWebRequest;
  Response: TWebResponse; var Handl ed: Bool ean);
var
  sw: TStringWri ter;
  xml w: TTrivial Xml Wri ter;
begin
  sw := TStringWri ter. Create;
  try
```

```

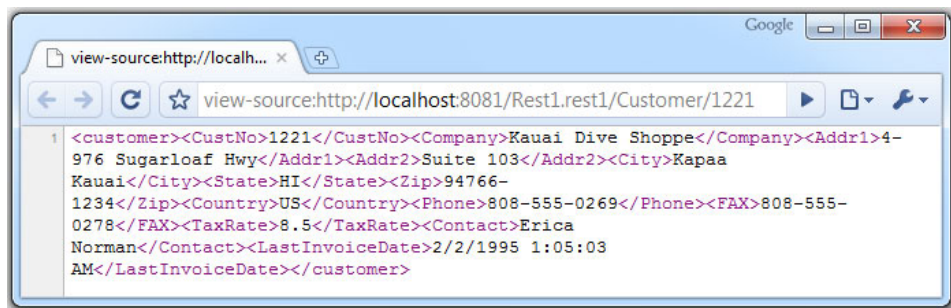
xmlw := TTrivialXmlWriter.Create (sw);
try
  xmlw.WriteStartElement('customers');
  cdsCustomers.First;
  while not cdsCustomers.Eof do
    begin
      xmlw.WriteStartElement('customer');
      xmlw.WriteStartElement('id');
      xmlw.WriteString(cdsCustomers['CustNo']);
      xmlw.WriteEndElement;
      xmlw.WriteStartElement('Company');
      xmlw.WriteString(cdsCustomers['company']);
      xmlw.WriteEndElement;
      xmlw.WriteEndElement;
      cdsCustomers.Next;
    end;
  xmlw.WriteEndElement;
finally
  xmlw.Free;
end;
Response.Content := sw.ToString;
Response.ContentType := 'text/xml';
finally
  sw.Free;
end;
end;

```

The last action returns data for a specific customer, an individual record. The PathInfo for this last action is different from the others. Rather than indicating a specific URL, in fact, it uses the notation `"/Customer/*"` to indicate any URL starting with that portion. The standard approach with REST, in fact is to use the URL path (and not its optional parameters) to refer to a given resource, in this case a customer. Following this style, we can use the following URL to refer to the first record:

■ `http://localhost:8081/Rest1.rest1/customer/1221`

This is the corresponding response in a web browser:



All we have to do in our last action (`acti onCustomer`) is to extract the last part of the URL and use it as a parameter:

```
const
  urlAction: AnsiString = '/Rest1.rest1/Customer/';
begin
  strCustId := Copy (Request.PathInfo,
    Length (urlAction) + 1, MaxInt);
```

Notice that the `PathInfo` property of the `Request` is an `AnsiString`, so we have to use the same string type of the constant part we want to remove from the path we receive¹¹².

Now that we have the customer id, we can write out each field of the given record, again using a `TTrivialXmlWriter` support object:

```
xmlw.WritelnStartElement('customer');
cdsCustomers.Locate('custno', strCustId, []);
for I := 0 to cdsCustomers.FieldCount - 1 do
begin
  xmlw.WritelnStartElement(cdsCustomers.Fields[I].FieldName);
  xmlw.WriteString(cdsCustomers.Fields[I].AsString);
  xmlw.WritelnEndElement;
end;
xmlw.WritelnEndElement;
```

Now that we have seen how the server can return the list of customers and individual customer records, let's focus on the corresponding Delphi client application. This time there is nothing specific to Delphi, so you could use other development tools for the client.

The code for processing the list is rather complex only because I'm parsing the resulting XML directly, rather than using an `XMLDocument` component and `DOM` or `XPath` (as in the `RSS Feeds` client demo, for example). I populate a list box from the resulting data with the displayed string having the *id=name* format:

```
procedure TForm3.btnCustListClick(Sender: TObject);
var
  strListCust, strId, strName: string;
  nPos: Integer;
  nInit, nEnd: Integer;
begin
  ListCust.Clear;
  strListCust := IdHTTP1.Get(BaseUrl + '/customers');
```

¹¹² Using ANSI strings for URL will not work properly with the new breed of “Non-Latin” URLs, recently introduced by ICANN (<http://www.icann.org/en/announcements/announcement-30oct09-en.ht>). As an example of a Chinese language URL you can use <http://例子.测试/首页> (or the easier-to-type version, <http://bit.ly/d2z7pp>).

```

nPos := Pos ('<customer><id>', strListCust);
while nPos > 0 do
begin
  nInit := nPos + 14; // Length <customer><id>
  nEnd := PosEx ('</id>', strListCust, nPos);
  strId := Copy (strListCust, nInit, nEnd - nInit);
  nPos := PosEx ('<company>', strListCust, nEnd);
  nInit := nPos + Length ('<company>');
  nEnd := PosEx ('</company>', strListCust, nPos);
  strName := Copy (strListCust, nInit, nEnd - nInit);
  ListCust.Items.Add (strId + '=' + strName);
  nPos := PosEx ('<customer><id>', strListCust, nEnd);
end;
end;

```

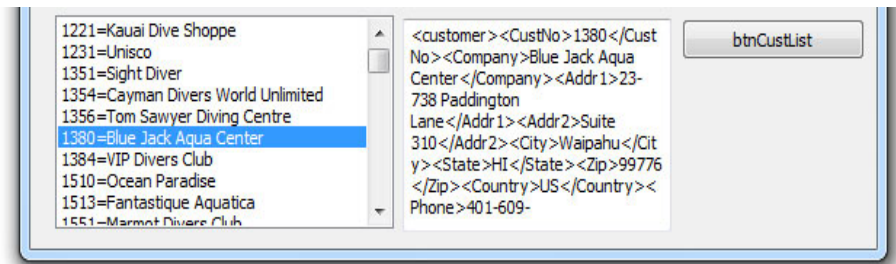
When you have the list of the customers, you can use the ID of the selected item to prepare the proper URL to grab its detailed data (which in this case is simply displayed *as is* on a Memo:

```

procedure TForm3. ListCustDbClick(Sender: TObject);
var
  strCust: string;
begin
  strCust := ListCust.Items.Names [ListCust.ItemIndex];
  MemoCust.Lines.Text :=
    IdHTTP1.Get(BaseUrl + '/customer/' + strCust);
end;

```

Here is the bottom part of the simple user interface of the Rest1Client application showing the list box with the customers and the raw XML data of the selected one:



Building a DataSnap REST Server

As we have seen in the previous section, you can build REST servers in Delphi using the WebBroker architecture. A second alternative is the use of the IdHTTPServer component. Further options are provided by third-party solutions. All of these options were already available in past versions of Delphi and are cer-

tainly still possible today. The focus of the rather long remaining part of the chapter, though, is the new REST support which is part of the DataSnap architecture in Delphi 2010.

To build a first simple DataSnap REST server¹¹³ in Delphi 2010 we can use the DataSnap Wizard, as we did in the last chapter for building other DataSnap servers. However, if you want to host your REST server as a web server, picking the DataSnap WebBroker Application will probably be your best choice. The DataSnap WebBroker architecture, in fact, lets you have more control over the HTTP requests coming in and lets you integrate your REST data within your WebBroker server.

As soon as you pick HTTP support in a DataSnap application or select a DataSnap WebBroker server, the resulting application will automatically include support for REST.

We have already seen in the last chapters which units are generated by the DataSnap WebBroker Wizard, here I'm only underlying a few specific elements worth considering for a REST server.

The web module generated by the Wizard is the core element of the WebBroker architecture. It can define multiple actions and has pre-processing and post-processing events for any HTTP request. The Wizard adds a DSHTTPWebDispatcher component to the web module:

```
object DSHTTPWebDispatcher1: TDSHTTPWebDispatcher
  RESTContext = 'rest'
  Server = DSServer1
  WebDispatcher.MethodType = mtAny
  WebDispatcher.PathInfo = 'datasnap*'
end
```

This component intercepts any request with a URL starting with *'datasnap'*, which are passed to the HTTP support of DataSnap. For requests starting with *'datasnap'* and indicating a *'rest'* path, the processing will be diverted to the built-in REST engine¹¹⁴. In other words, the requests with a *'datasnap/rest'* path are considered as REST requests.

113 A video of the development of this REST server application step-by-step is available on YouTube at <http://www.youtube.com/watch?v=TxFB8mjiGr8> and in my blog at http://blog.marcocantu.com/blog/rest_delphi_server_videos.html. As for the past videos mentioned in this chapter, I recorded this video for the Embarcadero white paper.

114 Notice that while you can change the *'rest'* element of the path with anything you like, you cannot remove it altogether. At the opposite, it seems impossible to change the *'datasnap'* portion. If you try to do so, you'll see an error when you try to connect.

The web module provides also a default HTTP response for any other action, simply returning some bare-bones HTML:

```
procedure TWebModule2.WebModule2DefaultHandlerAction(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<html><heading/><body>' +
    'DataSnap Server</body></html>';
end;
```

This action is configured at design-time as:

```
Actions = <
  item
    Default = True
    Name = 'DefaultHandler'
    PathInfo = '/'
    OnAction = WebModule2DefaultHandlerAction
end>
```

At the core of the application, like any DataSnap REST application, there is the server class, the class surfacing methods to be called remotely via REST. Notice that REST support is in fact focused only on server methods, and doesn't let you access the `IAppServer` interface exposed by the Dataset Provider components.

The skeleton class that gets generated is very simple, and depends on the fact that I asked for sample methods in the wizard. Here is the code, once again:

```
type
  TServerMethods1 = class(TDSServerModule)
  private
    { Private declarations }
  public
    function EchoString(Value: string): string;
end;
```

The `EchoString` method by default simply returns the passed parameter you are passing, but I've updated it slightly to repeat the tail of the string, as in a real "echo":

```
function TServerMethods1.EchoString(
  Value: string): string;
begin
  Result := Value + '...' +
    Copy(Value, 2, maxint) + '...' +
    Copy(Value, Length(Value) - 1, 2);
end;
```

Accessing the REST Server with a Browser

We can test this server to see if it works. After compiling and running the program, remember to run the Web App Debugger (available in Delphi's Tools menu), and start it using the corresponding button. The Web App Debugger runs on a specific port, by default 8081, so the URL will start with:

| `http://localhost:8081/`

Next in the URL comes the application name and the Web App Debugger class name (which in this case are identical), separated by a period:

| `FirstSimpleRestServer.FirstSimpleRestServer`

If you open the combined URL in a Web browser you can check if the Web App Debugger and the specific server are running. You should see something like:



This is the useless HTML returned by the program for the standard action. The next step is to use the specific URL for the only request our REST server can perform, calling the `EchoString` method of the `TServerMethods1` class using the 'rest' support of our 'datasnap' server. The URL is automatically combined by adding the REST server prefix (`/datasnap/rest`, by default), the class name, the method name, and the method parameters:

| `/datasnap/rest/TServerMethods1/EchoString/hello%20world`

In the URL the `%20` is just a replacement for a space, but you can actually type a space in your browser. Now the complete URL becomes:

| `http://localhost:8081/FirstSimpleRestServer.FirstSimpleRestServer/datasnap/rest/TServerMethods1/EchoString/hello%20world`

If you type it in a browser, you'll see the following JSON response:



The response you get when calling a server method is invariably a JSON object with a single pair called “*result*”. The value of this pair is always an array¹¹⁵, with the actual value returned by the method (a simple data type, an object, an actual array).

Notice that while doing this test we can use the Web App Debugger for figuring out the actual HTTP requests and responses being transferred. The page above is originated by a browser request:

```
GET /FirstSimpleRestServer.FirstSimpleRestServer/
  datasnap/rest/TServerMethods1/EchoString/hello%20world HTTP/1.1
Host: localhost:8081
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1;
  en-US; AppleWebKit/532.0 (KHTML, like Gecko)
  Chrome/3.0.195.27 Safari/532.0
Accept: application/xml,application/xhtml+xml,text/html;
  q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip, deflate, sdch
Cookie: LastProgid=
  FirstSimpleRestServer.FirstSimpleRestServer
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

This request produces the following complete HTTP response:

```
HTTP/1.1 200 200 OK
Connection: close
Content-Type: TEXT/HTML
Content-Length: 44

{"result":["hello world...ello world...ld"]}
```

As I mentioned earlier, the easy access to this low-level information can be a big bonus when debugging HTTP applications.

Returning Multiple Results

As mentioned earlier, the JSON result returned by the DataSnap REST support is invariably an array. This is necessary because as you can have multiple input parameters you can also have multiple output values, just use parameters passed by reference (var) or output parameters (out).

¹¹⁵ The result is returned in a JSON array structure as in more complex situations you might have further parameters passed by reference that the server method returns along with its result. I'll cover this in a second method of the current server.

As an example, I've added to the server a method with a parameter passed by reference and very simple code:

```
function TServerMethods1.TestParams(
    Value: string; var another: string): string;
begin
    Another := another + '*';
    Result := Value + another;
end;
```

Now if you call the server method from the browser with the URL:

```
http://localhost:8081/FirstSimpleRestServer.FirstSimpleRestServer/
datasnap/rest/TServerMethods1/TestParams/first/second
```

You'll get the following result, with an actual array of values, starting with the reference parameter(s) and ending with the function result:

```
{
  "result": [
    "second*",
    "firstsecond*"
  ]
}
```

There are further possible and complex scenarios, of course, but this explains in practice why the JSON result uses an array and shows that server method support is actually more sophisticated than it might appear at a first sight.

Calling the REST Server from a VCL Client

Now that we have built the server and made sure that it works, we can write a Delphi client application to test it. We can use two different approaches. One is to fall back to write a Delphi DataSnap client, using the specific transport layer provided by REST. But it won't make a lot of difference compared to using the HTTP or TCP transport layers of DataSnap.

The second option, which is the one I'm going to follow, is to create a custom REST client just like all of the various clients I built in the first part of this chapter. This means, you could use any other language for building the client application, as we are not relying on any specific Delphi support. To accomplish this simply create a standard Delphi VCL application, add an IdHTTP component to it to perform the actual REST request, an edit box for the input, and a button with the code:

```
const
    strServerUrl = 'http://localhost:8081/' +
        'FirstSimpleRestServer.FirstSimpleRestServer/';
```

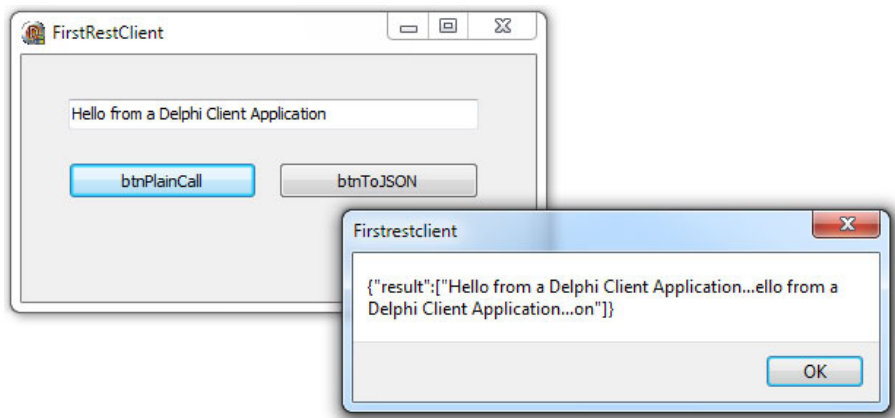
```

    strMethodUrl = 'datasnap/rest/TServerMethods1/EchoString/';

procedure TFormFirstRestClient.btnPlainCallClick(Sender: TObject);
var
    strParam: string;
begin
    strParam := edInput.Text;
    ShowMessage (IdHTTP1.Get(strServerUrl +
        strMethodUrl + strParam));
end;

```

This call builds a proper URL by concatenating the server address, the relative path to reach the given method with the REST server, and the only parameter. The call results in the following output:



Now what is more interesting is to extract the actual information from the JSON data structure returned by the server. We could use a manual approach, but I'd rather take advantage of the JSON support that is available in Delphi 2010 and made available through the DBXJSON unit.

The JSON data that our server returns is a string, but the REST server support creates an object with a named value (or “pair”), and places the actual value in an array. That's why after parsing the result of the HTTP into a JSON data structure, we need to navigate from the object to the pair it contains and from the pair to the single element array it holds:

```

procedure TFormFirstRestClient.btnToJSONClick(
    Sender: TObject);
var
    strParam, strHttpResult, strResult: string;
    jValue: TJSONValue;
    jObj: TJSONObject;
    jPair: TJSONPair;
    jArray: TJSONArray;

```



```

begin
  strParam := edInput.Text;
  strHttpResult := IdHTTP1.Get(strServerUrl +
    strMethodUrl + strParam);
  jValue := TJSONObject.ParseJSONValue(
    TEncoding.ASCII.GetBytes(strHttpResult), 0);
  if not Assigned(jValue) then
  begin
    ShowMessage('Error in parsing ' + strHttpResult);
    Exit;
  end;

  try
    jObj := jValue as TJSONObject;
    // get the first and only JSON pair
    jPair := jObj.Get(0);
    // pair value is an array
    jArray := jPair.JsonValue as TJSONArray;
    // get the first and only element of array
    strResult := jArray.Get(0).Value;
    ShowMessage('The response is: ' + strResult);
  finally
    jObj.Free;
  end;
end;

```

Again, the complexity is due to the data structure returned by the server, as in other circumstances it would be much easier to parse the resulting JSON and access to it.

Calling the REST Server From a jQuery Client

If all you need is to pass object data from a server side Delphi application to another one, there could be many alternatives to using JSON. This choice makes a lot of sense when you want to call the Delphi compiled server from a JavaScript application running in the browser. This case is quite relevant because AJAX (Asynchronous JavaScript calls done in the Web browser) was and still is one of the driving forces behind the adoption of REST. Calling a corresponding SOAP server from a Browser based program is incredibly more complicated.

So, how can we create an application mimicking the client I just wrote but running in the Web browser. I could have used many different approaches and libraries, but my preference at this time is to use jQuery, an incredibly powerful open source JavaScript library available at:

http://jquery.com

I don't have time to delve into jQuery and its usage in this chapter, but I'll at least try to explain the jQuery code behind this specific example. First of all, the HTML page includes jQuery and its JSON support:

```
<head>
  <title>jQuery and Delphi 2010 REST</title>
  <script src="http://jqueryjs.googlecode.com/
    files/jquery-1.3.2.min.js"
    type="text/javascript"></script>
  <script src="http://jquery-json.googlecode.com/
    files/jquery.json-2.2.min.js"
    type="text/javascript"></script>
</head>116
```

Second, the page has a very simple user interface, with some text, an input field and a button (without any sophisticated CSS and added graphics, as I really wanted to keep this focused):

```
<body>
  <h1>jQuery and Delphi 2010 REST</h1>

  <p>This example demonstrates basic use of jQuery calling
    a barebones Delphi 2010 REST server.</p>
  <p>Insert the text to "Echo": <br/>

  <input type="text" id="inputText" size="50"
    value="This is a message from jQuery">
  <br/>
  <input type="button" value="Echo" id="buttonEcho">

  <div id="result">Result goes here: </div>
</body>
```

If this is the skeleton, let us now look at the actual JavaScript code. What we have to do is add an event handler to the button, read the input text, make the REST call, and finally display the result. I'm going to use the simplest of jQuery selectors, based on the objects ID, to access to the page objects as in:

```
| $("#inputText")
```

This returns a jQuery object wrapping the input text DOM element. To define an event handler we can pass an anonymous method parameter to the `click()` function of the button. Two more calls are the REST call itself (using the global `getJSON`) and the `html()` call to add the result to the HTML of the element.

This is the complete code at the heart of this demo, a very compact but not exactly readable JavaScript snippet:

¹¹⁶ I haven't made any attempt to optimize the HTML or JavaScript so that the examples remain as clear as possible.

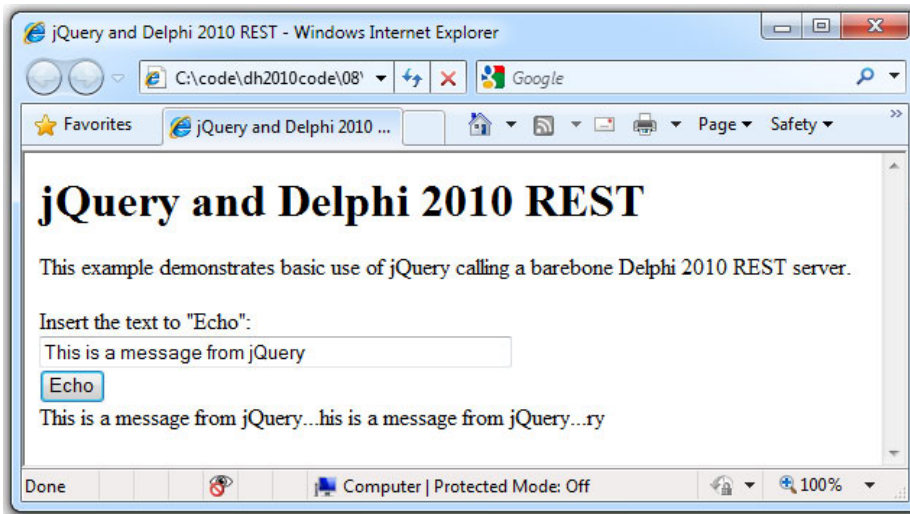
```

$(document).ready(function() {
    $("#buttonEcho").click(function(e) {
        $.getJSON("http://localhost:8081/"
            "FirstSimpleRestServer.FirstSimpleRestServer/"
            "datasnap/rest/TServerMethods1/EchoString/" +
            $("#inputText").val(),
            function(data) {
                $("#result").html(data.result.join(' '));
            });
    });
});

```

Just by opening an HTML file with the given code you can call the custom server, but only if the browser permission settings allow an AJAX call from a local file to a local REST server. In general, most browsers will only let you call REST servers on the same site originating the HTML page.

In any case, Internet Explorer seems to work fine on this local file, after enabling local scripts and asking for limited security (available since the file is on the local machine, see the icons in the status bar):



On other web browsers, you need to let the server return both the HTML page and the REST data, which is not a terribly big deal as our REST server is indeed a web server. So, all I had to do for a “server side” solution (as far as the web browser is concerned) is to add an action to the web server module (which I hooked to the “/file” URL) and return the HTML file from it:

```

procedure TWebModule2.WebModule2WebActionItem1Action(
    Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);

```

```

var
  strRead: TStreamReader;
begin
  strRead := TStreamReader.Create('jRestClient.html');
  try
    Response.Content := strRead.ReadToEnd;
  finally
    strRead.Free;
  end;
end;

```

Now we can refer to given server page with the /file URL, get the file with the JavaScript code, and let it call our REST server:



The difference between this and the previous image is not just that I'm using a different browser, but that I'm pointing to a different URL. Rather than loading a file, in this second case I'm using the server side REST application as a full Web server, returning the HTML used for calling the same server via AJAX.

Returning and Updating Objects with REST HTTP Methods

Now that we have explored the development of a very simple REST server with Delphi 2010 DataSnap support, it is time to try to figure out the actual code we can write on the server to make it more powerful. As we have seen, the server returns JSON data, converting the result of your functions to this format. We can pass an object as result and have its data converted. However, in most

practical situations, it would be better to take full control and create specific JSON objects on the server side and return them. That is going to be one of the goals of our next project.

The same project will also show how to process other HTTP methods beside the Get method, letting us not only retrieve but also modify a server side object from a simple Browser-based client written in JavaScript. Finally, in doing this we'll focus on URL management and figure out how to make them nicer and more flexible.

Returning JSON Objects and Values

For this second project I've used the DataSnap WebBroker Wizard, picked the Web App Debugger architecture (again), and decided to use a `TPersistent` base class, as I don't need a data module as target class for the REST calls. As you need specific RTTI support, the rule is to inherit (at least) from `TPersistent` and mark the class with the `$METHODINFO` directive, as in the following generated code:

```
{ $METHODINFO ON }
type
  TObjectsRest = class(TPersistent)
  public
    function PlainData (name: string): TJSONValue;
    function DataMarshal (name: string): TJSONObject;
  end;
{ $METHODINFO OFF }
```

As you can see I've added a couple of functions to the class for returning either a value or a full object¹¹⁷. Later I'll add other methods to the class.

The data structure behind this application is a list of objects of a custom type (which could have been written in a more object-oriented way, but I wanted to keep it simple for the sake of the example):

```
type
  TMyData = class (TPersistent)
  public
    Name: String;
    Value: Integer;
  public
    constructor Create (const aName: string);
  end;
```

¹¹⁷ More correctly, all the data values of a full object.

The objects are kept in a dictionary, implemented using the generic container class `TObjectDictionary<TKey, TValue>` defined in the `Generics.Collections` unit since Delphi 2009. This global object is initialized when the program starts with the addition of a couple of predefined objects. Notice that I use a specific `AddToDictionary` procedure to add the objects, to make sure the object name is in sync with the dictionary key and it has a random value if none is provided:

```
var
  DataDict: TObjectDictionary<string, TMyData>;

procedure AddToDictionary (const aName: string;
  nVal: Integer = -1);
var
  md: TMyData;
begin
  md := TMyData.Create (aName);
  if nVal <> -1 then
    md.Value := nVal;
  DataDict.Add(aName, md);
end;

Initialization
  DataDict := TObjectDictionary<string, TMyData>.Create;
  AddToDictionary('Sample');
```

Having this data structure in place, we can now focus on the first two sample methods used to return the JSON values. The first returns the value of the given object (picking a default one if no parameter is passed to the function):

```
function TObjectRest.PlaInData(name: string): TJSONValue;
begin
  if Name = '' then
    name := 'Sample'; // default
  Result := TJSONNumber.Create(DataDict[name].Value);
end;
```

If we use an URL with or without the parameter (as in the following two lines):

```
/datasnap/rest/TObjectRest/PlaInData/Test
/datasnap/rest/TObjectRest/PlaInData
```

we will still get a JSON response, either for the specific object or a default one:

```
{"result": [8978]}
```

What if we want to return a complete object rather than a specific value? Our REST server cannot return a `TObject` value, as the system has no way to convert it automatically, but it can indeed use the new JSON marshaling support for converting an existing object to the JSON format:

```
function TObjectRest.DataMarshal (name: string): TJSONObject;
var
  jMarshal: TJSONMarshal;
```

```

begin
  jMarshal := TJSONMarshal.Create(TJSONConverter.Create);
  Result := jMarshal.Marshal(DataDict[name])
    as TJSONObject;
end;

```

This approach is mostly useful when you need to recreate the object in the Delphi client application, while it is not particularly handy in the case where the client is written in another language. The resulting JSON looks a little ugly:

```

{
  "result": [{
    "type": "ObjectsRestServer_Classes.TMyData",
    "id": 1,
    "fields": {
      "Name": "Test",
      "Value": 8068}
    }]
}

```

So, what would be the best option to return a JSON object? I think it would be to create one on the server side, using the support classes. This is what I've done in the MyData function:

```

function TObjectsRest.MyData(name: string): TJSONObject;
var
  md: TMyData;
begin
  md := DataDict[name];
  Result := TJSONObject.Create;
  Result.AddPair(TJSONPair.Create('Name', md.Name));
  Result.AddPair(TJSONPair.Create(
    'Value', TJSONNumber.Create(md.Value)));
end;

```

As you can see I've created a TJSONObject and added two pairs (or properties) for the name and the value. I could have used a dynamic name (that is, used the name for the name part of the pair), but this would have made it harder to retrieve the data on the client side. The result of this code should look like the following cleaner JSON code:

```

{
  "result": [{
    "Name": "Test",
    "Value": 8068
  }]
}

```

Listing Objects with a TJJSONArray

Now having a list of objects, you might well need to access the list of objects. Having a list with the names only (and no data) will be useful when building a client side user interface.

For returning a list you can use a TJJSONArray, which in this case will be an array of strings I create using an enumerator on the Keys of the dictionary:

```
function TObjectRest.List: TJJSONArray;
var
  str: string;
begin
  Result := TJJSONArray.Create;
  for str in DataDict.Keys do
  begin
    Result.Add(str);
  end;
end;
```

The result of this call is an array in JSON format, which in turned is passed (as usual) in an array called result (hence the double nested array notation):

```
{
  "result": [
    ["Test", "Sample"]
  ]
}
```

Now that we have a way to return a list of values and fetch the data of each individual element, we can start building a user interface.

Sending the List to the jQuery Web Client at Start-up

Rather than having to build the initial HTML with the list of values, to let the user pick one, we can fully exploit the AJAX model.

The page on start up will have no data at all, only the HTML elements and the JavaScript code. As soon as the page is loaded, even without user intervention, it will ask the server for the actual data and populate the user interface.

As an example, on start up the program shows the value of the Sample object, using the following HTML elements and AJAX call (executed when document is ready, that is the DOM has finished loading):

```
<div>Sample: <span id="sample"></span></div>
```



```

<scri pt>
    var baseUrl =
        "/ObjectsRestServer.RestObjects/dataSnap" +
        "/rest/TObjectsRest/";

    $(document).ready(function() {
        $.getJSON(baseUrl + "MyData/Sample",
            function(data) {
                strResult = data.result[0].Value;
                $("#sample").html(strResult);
            });
    });
</scri pt>

```

The AJAX call to MyData passes the object name as a further URL parameter and extracts from the result array the property/pair called Value, showing it in an empty span HTML element. Something similar (but somewhat more complex) happens for the list. Again, there is an AJAX call, but this time we have to build the resulting HTML. The operation is performed in a separate refreshList function called both automatically at start-up and manually by the user:

```

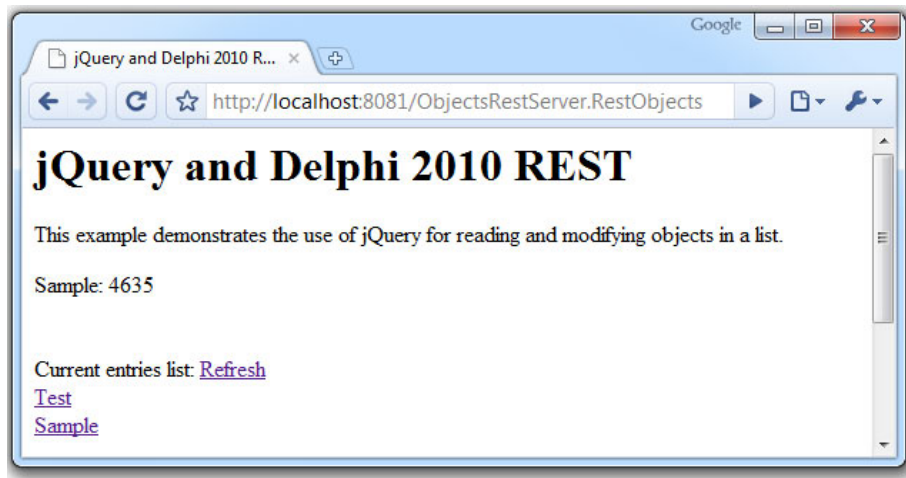
<div>Current entries list:
    <a href="#" id="refresh">Refresh</a>
    <span id="list"></span></div>

function refreshList()
{
    $.getJSON(baseUrl + "list",
        function(data) {
            var thearray = data.result[0];
            var ratingMarkup = ["<br>"];
            for (var i=0; i < thearray.length; i++) {
                ratingMarkup = ratingMarkup +
                    "<a href='#'>" + thearray[i] + "</a><br>";
            }
            $("#list").html(ratingMarkup);
        });
};

```

The code uses a for loop to scan the resulting array. I could have used the \$.each enumeration mechanism of jQuery, but this would have made the code more complex to read. The for loop creates the HTML, which is later displayed in the span place-holder with the given ID. In the next page you can see the sample output with the value of the Sample object (the code shown earlier) plus the list of the values returned in the JSON array.

As I mentioned earlier, the refreshList function is called at start-up (in the ready event handler) and also connected with a corresponding link, so that the



user can later refresh the data of the list without having to refresh the entire HTML page:

```
$(document).ready(function() {
    refreshList();
    $("#refresh").click(function(e) {
        refreshList();
    });
});
```

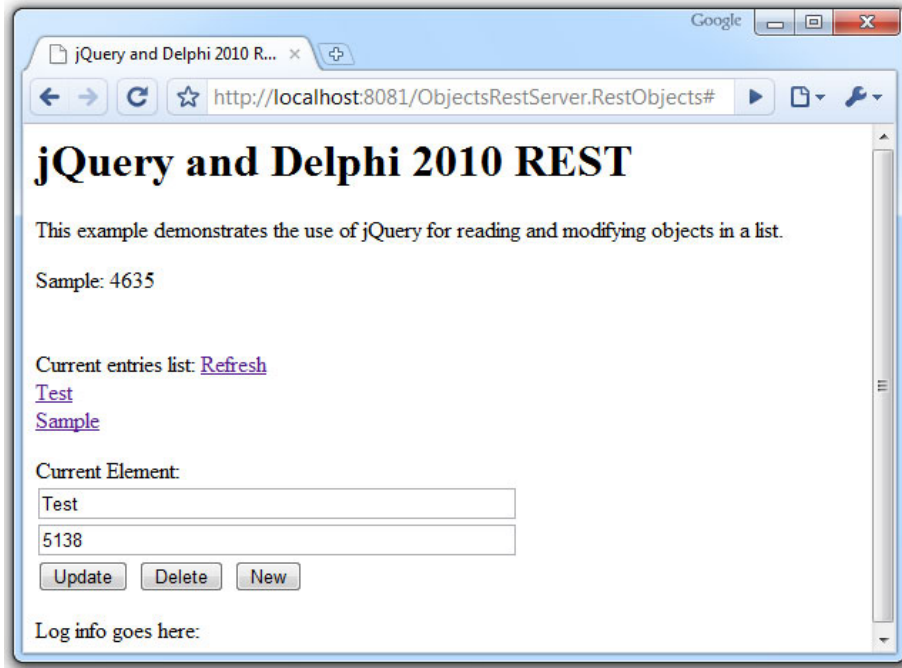
There is actually a little more to the code generation. As soon as we have the HTML for the list, which is a list of links, we need to hook code to those links so that selecting each entry of the list the client application will load the corresponding server side object. The user interface for the object data is made of two input boxes, which will later use also for manipulating the object data. The behavior is added to each anchor within the list container.

```
$("#list").find("a").click(function(e) {
    var wasClicked = $(this);
    $.getJSON(baseUrl + "MyData/" + $(this).html(),
        function(data) {
            strResult = data.result[0].Value;
            $("#inputName").val(wasClicked.html());
            $("#inputValue").val(strResult);
        });
});
```

Notice the use of the `$(this)` call, which is more or less the Sender parameter for a Delphi event. Its text is the html content of the element that was clicked, which is the name of the element we have to pass to the server in the URL, with the expression:

```
baseUrl + "MyData/" + $(this).html()
```

Now with this code in place we can see the effect of clicking on one of the elements of the list: A further AJAX call will reach our server asking for a given value, and the returned value is displayed in two input text boxes:



As you can see above, the program let us retrieve a value, but has also three buttons to perform the most common operations (the so called CRUD interface – Create, Read, Update, Delete). This is supported in HTML using the 4 code HTML methods, respectively PUT, GET, POST, and DELETE. How these are supported by a Delphi 2010 REST server is the subject of the next section.

HTTP Methods: POST, PUT, and DELETE

Up to now we have seen only how to get data from our REST server, but what about updating it? The generally agreed idea in REST is to avoid using specific URLs for identifying the operations, but use an URL only to identify server side objects (like `MyData/Sample` in our case) and use the HTTP methods to indicate what do to.

Now if Delphi's REST support simply mapped URLs to methods, we would have been out of luck. Instead, it maps URLs plus the HTTP method to meth-

ods, using a rather simple scheme: the name of the operation is prepended to the method name, using the following mapping:

GET	get (default, can be omitted)
POST	update
PUT	accept
DELETE	cancel

You can customize these mappings by dealing with the four corresponding event handlers of the `DSHTTPWebDispatcher` component. If we decide to go with the standard naming rules, to support the various operations we need to define our server class as:

```
type
  TObjectsRest = class(TPersistent)
  public
    function List: TJSONArray;
    function MyData (name: string): TJSONObject;
    procedure updateMyData (name, value: string);
    procedure cancel MyData (name: string);
    procedure acceptMyData (name, value: string);
  end;
```

To get or delete an element we only need the name, while to create or update it we need a second parameter with the data.

The implementation of the three new methods is rather simple and direct, also because they don't need to return a value (needless to say I should have checked that parameters are not empty and that the server side object really exists in the container):

```
procedure TObjectsRest.updateMyData (name, value: string);
begin
  DataDict[name].Value := StrToIntDef (Value, 0);
end;

procedure TObjectsRest.cancel MyData(name: string);
begin
  DataDict.Remove(name);
end;

procedure TObjectsRest.acceptMyData(name, value: string);
begin
  AddToDictionary (name, StrToIntDef (Value, 0));
end;
```

Editing Data with jQuery

Now that we have the CRUD operations available on the REST server, we can complete our JavaScript client application with the code of the three editing buttons (the image of the browser-based user interface was shown earlier).

While jQuery has specific support for the get operation (with different versions, including the JSON-specific one we have used earlier) and some support for post operations, for the other HTTP methods you have to use the lower level (and slightly more complex) \$. ajax call. This call has as a parameter a list of paired values, up to a dozen that are possible. The more relevant parameters are the type and the URL, while data lets you pass further POST parameters.

Posting our update is rather simple, and we can provide the data to our REST server using the URL:

```
$("#buttonUpdate").click(function(e) {
    $.ajax({
        type: "POST",
        url: baseUrl + "MyData/" +
            $("#inputName").val() + "/" +
            $("#inputValue").val(),
        success: function(msg){
            $("#log").html(msg);
        }
    });
});
```

Deleting is equally simple, as we need to create the URL with the reference to the object we want to remove:

```
$("#buttonDelete").click(function(e) {
    $.ajax({
        type: "DELETE",
        url: baseUrl + "MyData/" +
            $("#inputName").val(),
        success: function(msg){
            $("#log").html(msg);
        }
    });
});
```

It took me some more time to figure out how to implement PUT, as if you don't provide any data, some browsers (notably Chrome) will post the data as "undefined" and this will make the HTTP input parsing of the REST server crash with an error.

As we need to pass information (and we cannot pass more parameters than the server requires, which will be equally flagged as an error), what we can do is replace one of the URL elements with a corresponding data element:

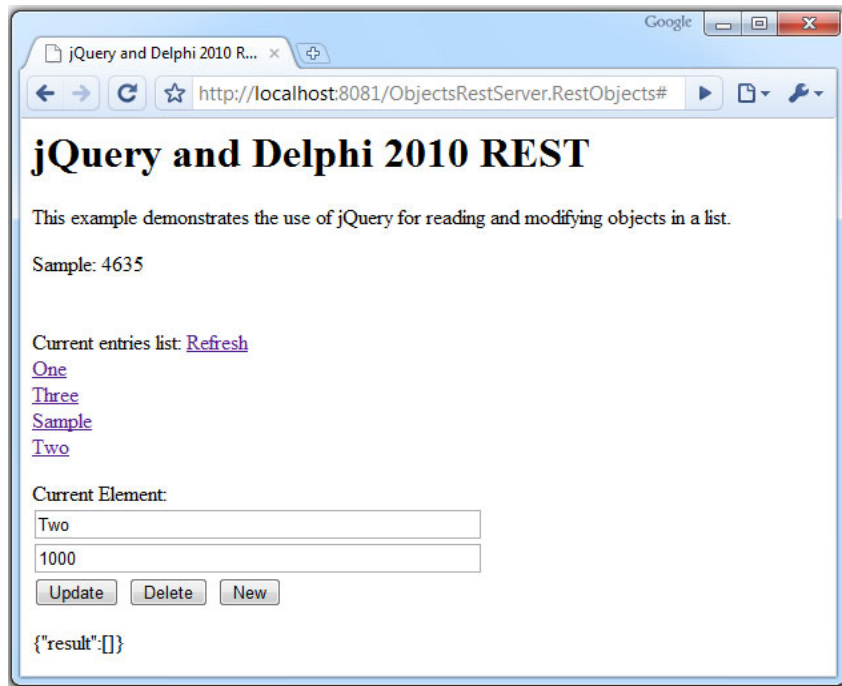
```

$("#buttonNew").click(function(e) {
    $.ajax({
        type: 'PUT',
        data: $("#inputValue").val(),
        url: baseUrl + "MyData/" +
            $("#inputName").val(),
        success: function(msg) { $("#log").html(msg); }
    });
});

```

Notice that jQuery documentation specifically warns against using PUT in browsers, as you might get mixed results. That might as well be the reason for which a number of REST services (including those from Microsoft) tend to use POST for both updating and creating server side objects. I prefer keeping the two concepts separate, for clarity and consistency, whenever possible.

With the three extra methods added to our class and the proper AJAX calls, we now have an example with a complete Browser-based user interface for creating and editing objects in our REST server. Here I've created a few objects:



Building a Database Oriented REST Server

If the original idea behind DataSnap focused on moving data tables from a middle-tier server to a client application, it might be quite odd at first to realize that you cannot return a dataset from a REST server written in Delphi 2010. Well, you cannot return it directly or as easily as you return its XML representation, but you can create a JSON result with all of the data of a Delphi dataset. That's the focus my last example.

The program is quite bare bones, as all it does is return the data of a complete Dataset, with no metadata. It could be extended in several ways and lacks a polished user interface, but should get you started. The server class has only one method, returning an entire dataset in a JSON array:

```
function TServerData.Data: TJSONArray;
var
  jRecord: TJSONObject;
  I: Integer;
begin
  ClientDataSet1.Open;
  Result := TJSONArray.Create;

  while not ClientDataSet1.EOF do
    begin
      jRecord := TJSONObject.Create;
      for I := 0 to ClientDataSet1.FieldCount - 1 do
        jRecord.AddPair(
          ClientDataSet1.Fields[I].FieldName,
          TJSONString.Create(ClientDataSet1.Fields[I].AsString));
      Result.AddElement(jRecord);
      ClientDataSet1.Next;
    end;
end;
```

This method is invoked by the client application after loading the page, building an HTML table dynamically, with the following jQuery code (you should have become familiar with the coding style by now):

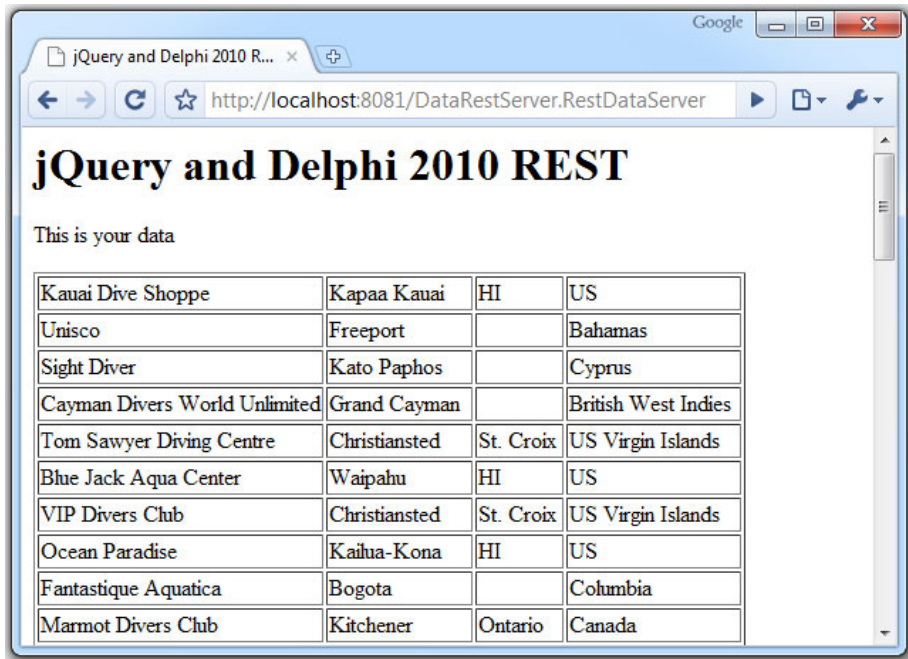
```
$(document).ready(function() {
  $.getJSON("/DataRestServer.RestDataServer/datasnap" +
    "/rest/TServerData/Data",
    function(data) {
      var thearray = data.result[0];
      var ratingMarkup = "<table border='1'>";
      for (var i=0; i < thearray.length; i++) {
        ratingMarkup = ratingMarkup + "<tr><td>" +
```

```

        thearray[i].Company + "</td><td>" +
        thearray[i].City + "</td><td>" +
        thearray[i].State + "</td><td>" +
        thearray[i].Country + "</td></tr>";
    }
    ratingMarkup = ratingMarkup + "</table>";
    $("#result").html(ratingMarkup);
});
});

```

The bare-bones result is visible below in a web browser:



Can we improve it a little bit? The final version of the program adds some metadata support to improve the final output.

On the server side, there is a second function returning an array of field names from the dataset field definitions:

```

function TServerData.Meta: TJSONArray;
var
    jRecord: TJSONObject;
    I: Integer;
begin
    ClientDataSet1.Open;
    Result := TJSONArray.Create;
    for I := 0 to ClientDataSet1.FieldDefs.Count - 1 do
        Result.Add(ClientDataSet1.FieldDefs[I].Name);
    end;
end;

```


The client-side JavaScript has been expanded with a second call to the REST server to get the metadata:

```
$.getJSON("/DataRestServer.RestDataServer/datasnap/" +
    "rest/TServerData/Meta",
    function(data) {
        theMetaArray = data.result[0];
```

This information is used to create the table header and to access the object properties dynamically, with the notation:

```
object["propertyname"]
```

In our case the existing code used to access to an object with the property symbol:

```
thearray[i].Company
```

becomes the following code that reads the property by name, using the name of the field stored in the metadata:

```
thearray[i][theMetaArray[j]].
```

This is the complete JavaScript code used to create the HTML markup¹¹⁸:

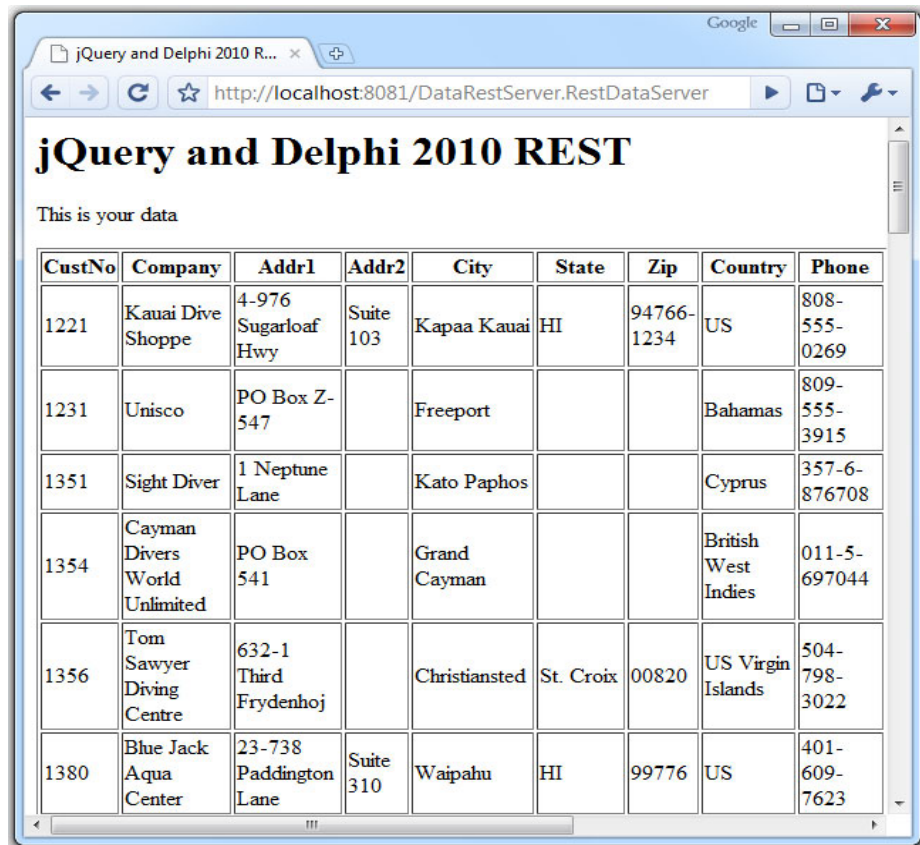
```
var ratingMarkup = "<table border='1'><tr>";

// header
for (var j=0; j < theMetaArray.length; j++) {
    ratingMarkup = ratingMarkup + "<th>" +
        theMetaArray[j] + "</th>";
};
ratingMarkup = ratingMarkup + "</tr>";

// content
for (var i=0; i < thearray.length; i++) {
    ratingMarkup = ratingMarkup + "<tr>";
    for (var j=0; j < theMetaArray.length; j++) {
        ratingMarkup = ratingMarkup + "<td>" +
            thearray[i][theMetaArray[j]] + "</td>";
    };
    ratingMarkup = ratingMarkup + "</tr>";
}
ratingMarkup = ratingMarkup + "</table>";
```

The output of this extended version becomes slightly nicer (and more flexible):

¹¹⁸ For concatenating script in JavaScript it is more common, and shorter, to use the += operator, but here I've kept the code more "Pascal-like", to make it easier for the average reader of this book.



CustNo	Company	Addr1	Addr2	City	State	Zip	Country	Phone
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa Kauai	HI	94766-1234	US	808-555-0269
1231	Unisco	PO Box Z-547		Freeport			Bahamas	809-555-3915
1351	Sight Diver	1 Neptune Lane		Kato Paphos			Cyprus	357-6-876708
1354	Cayman Divers World Unlimited	PO Box 541		Grand Cayman			British West Indies	011-5-697044
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj		Christiansted	St. Croix	00820	US Virgin Islands	504-798-3022
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310	Waipahu	HI	99776	US	401-609-7623

Again, this serves only as a starting point, and I didn't use any of the jQuery plug-ins, which would add significant capabilities to HTML tables, turning them into powerful user interface grids, with sorting, filtering, and editing capabilities.

REST Server Alternatives

After an introduction on building a REST server with the plain WebBroker architecture, in this chapter I've delved into the development of REST applications in Delphi 2010 using the specific DataSnap support. With this support, as we have seen, you can map server methods to URLs and handle the various

HTTP methods. I have also introduced the development of simple JavaScript clients using the jQuery library.

Now the question becomes, is this the suggested way to implement a REST server in Delphi? I think the current DataSnap REST mapping has a few limitations, including a very rigid first part of the URL (a thing that can be worked around), the inability to combine HTTP query parameters with URL elements (the URL should refer to a resource in REST, but any additional parameter should be passed as an HTTP parameter), and the availability of only one return type, JSON (which is quite handy, but still having alternatives would be better).

On the positive side, this architecture integrates very well with DataSnap and server side methods (making it well suited to provide a back-end to both Delphi clients and browser-based applications with a single server side technology), it is quite simple to use (and you don't have to work with any low-level technology, do your own URL parsing, and so on), and let's you stay focused on Delphi code for your servers.

I do have a lot of investment in server side web and REST applications written in Delphi, and in the recent years I've started playing with and introducing at conferences a Delphi Web Application REST Framework¹¹⁹ (that is, DWARF), which at this time is still not publicly available... simply because it is too sketchy and unfinished to be published. I've seen other ongoing efforts to clone Rails in Delphi and offer other REST server architectures.

I think that if you want to build a very large REST application architecture you should roll out your own technology or use one of these prototypical architectures. For a small to medium size effort, on the other hand, you can probably benefit from the native DataSnap support. This is particularly true if you want the REST access to go along with a native DataSnap HTTP access focused on Delphi clients. In this case, in fact, you get the REST support for free as you built your multi-tier Delphi architecture.

¹¹⁹ I mentioned my framework only once in my blog, but without providing much detail, as you can see at http://blog.marcocantu.com/blog/web_dev_stacks_delphi.html

What's Next

As this is the end of the book there is no introduction to the following chapters, but only a reminder of where to find extended information. As mentioned in the introduction, refer to the book web page, my home page or my blog:

<http://www.marcocantu.com/dh2010>

<http://www.marcocantu.com>

<http://blog.marcocantu.com>

It is very likely you'll find corrections and updates to the book, and possibly additional chapters in PDF format which you could buy online.

I hope you have enjoyed the book and it will help you use the extended power of Delphi 2010 in a more effective way. If this is the case (but also if you found errors, omissions, or just disliked the book) an email or online message of any form is always welcome.

On to the next Delphi!

Index

42.....	115	Background Compilation.....	38
90 degree rotation.....	155	Bamboo Touch.....	166
Acer.....	166	Barry Kelly.....	78
ActionList.....	158, 172, 180	BeginDraw.....	146
ActionManager.....	172	Berry Kelly.....	74
Actions.....	222	blog.....	10
ActivateKeyboardLayout.....	185	Bob Swart.....	203
AddGesture.....	178	Borland.....	5, 208
ADO 2.8.....	208	Borland Database Engine.....	206
ADODataset.....	208	breakpoint.....	50
Aero Glass.....	143	Build All.....	35
AJAX.....	288, 290, 295	C#.....	111
Alexey Barkovoy.....	132	Callback.....	249
Allen Bauer.....	110	Callbacks.....	247
Alternative Document Object Model.....	256	Canvas.....	147
Amazon.....	7	CaptionFont.....	157
Andreano Lanusse.....	39	Cary Jensen.....	10
Andreas Hausladen.....	39, 207	CategoryButtons.....	157
anonymous method.....	121	CGI.....	222
Apache.....	222	CheckBox.....	157
Apple Inc.....	164	CheckWin32Version.....	126
Application.....	127	Chris Bensen.....	9, 158, 184, 186, 188, 190
as.....	104	Chris Hesik.....	50
AsLargeInt.....	200	Class Constructors.....	106p.
AsSingle.....	200	class destructor.....	106
Attributes.....	82	ClientDataSet.....	37, 180p., 227, 257, 277
Author.....	9	CloseTouchInputHandle.....	187

310 - Index

Code Completion.....	33	SMETHODINFO.....	292
Code Formatter.....	31	SRTTI.....	66, 70
CodeGear.....	5	\$SCOPEENUMS.....	111
CommandType.....	220	\$StrongLinkTypes.....	70
Compiler.....	103	\$WeakLinkRTTI.....	70
compiler version.....	104	Directory Groups.....	30
Component Editor Pane.....	37	DirectWrite.....	151
Component Toolbar.....	39	DirectX.....	132, 143
conflicting gestures.....	177	Document Object Model.....	255
CPU view.....	50	DoubleBuffered.....	128
CreateComObject.....	137	Douglas Adams.....	115
CreateTextFormat.....	152	DrawBitmap.....	144
cross-platform.....	113	DrawingStyle.....	202p.
Ctrl + <period>.....	23	DrawText.....	144, 152
Custom Gestures.....	174	DSAuthUser.....	219
D2D1ColorF.....	147	DSHTTTPService.....	213, 215
Daniel Wischniewski.....	136	DSHTTTPServiceAuthenticationManager.....	218
Daniele Teti.....	8, 232	DSHTTTPWebDispatcher.....	224p., 229, 232, 282, 299
Data Explorer.....	216	DSPProviderConnection.....	212, 227
DataSnap... 211, 214, 216, 218, 221, 223, 228, 250, 281, 292, 306		DSServer.....	211, 224
DataSnap Filters Compendium.....	232	DSServerClass.....	211, 215p., 224p.
DataSource.....	180	DSTCPServerTransport.....	212, 230
dbExpress.....	199, 208	DWriteFactory.....	151
DBGGrid.....	157, 179pp., 202, 204, 227	editor bookmarks.....	28
dbxdrivers.ini.....	210	EditorMode.....	204
Debug Inspector.....	49	EExternalException.....	110
Debugger.....	47	efficient code.....	113
Debugger Visualizers.....	53	Elastic Margin.....	192
Dee Elling.....	20	Electronic Software Delivery.....	21
DefaultFont.....	127	email.....	10
delayed.....	110, 133p.	Embarcadero Technologies.....	5, 209
Delayed Loading.....	109	EndDraw.....	146
DELETE.....	298p.	Essential Pascal.....	6
Dell.....	166	Event Log.....	49
Delphi 2007 Handbook.....	6, 126, 129p., 152	Example.....	
Delphi 2009 Handbook.....	7, 71, 94, 111, 211, 213, 240, 247	ClassCtor.....	106
Delphi Developer Days.....	10	CustomGestureTest.....	175, 178
Description Pane.....	36	D2DGradients.....	149
Desktop Windows Manager.....	134	D2DIntro.....	145
Desktop Windows Manager API.....	134	DataGestures.....	182
Dieter Kohler.....	256	DbxMulti2010.....	202, 209
Direct2D.....	145, 148, 151, 195	DebugVisual.....	54, 56
Direct2DCanvas.....	149	DelayedLoading.....	110
Directive.....		DSnapFilterDemo.....	229, 231
		DSnapHttpConsole.....	214

DSnapJson.....	243	Facebook page.....	10
DsnapMethodsCallback.....	247	FileOpenDialog.....	128, 140
DSnapWebAppDebug.....	225	FileSaveDialog.....	128
DWrite.....	151	Filter Wild Cards.....	24
EditGestures.....	172	Filtering.....	228
FileAccess.....	130	finalization.....	106
First3Tier2009.....	213	Find in Files.....	30
GenericClassCtor.....	108	Firebird.....	208p., 225
GeoLocation.....	266	Flip 3D.....	127, 129
Gestures01.....	168	Focused.....	182
GetOSVersion.....	126	Format Source.....	32
InertiaBall.....	191	Fredrik Haglund.....	130
IoFilesInFolder.....	118	Gallery.....	42
JsonMarshal.....	238, 240	Generate DataSnap client classes.....	221
JsonTests.....	235, 237	Generic Classes.....	107
KeyboardTest.....	184	generics.....	114
LargeXml.....	257	Geocoding.....	266
MiniPack.....	69	Gesture Manager.....	168
MiniSize.....	68	GestureListView.....	174, 178
MoveIP.....	48	GestureManager.....	168, 173, 175, 177p., 180
NamedThreads.....	51p.	GesturePreview.....	174, 178p.
ObjFromIntf.....	104	GestureProvider.....	179
ReadOnlyRecord.....	112	GestureRecorder.....	174
Rest1.....	275	Gestures.....	166, 169
Rest1Client.....	276p., 281	GET.....	262, 298p.
RssClient.....	264	GetDirectories.....	119
RtlLists.....	115p.	GetFiles.....	119, 122
RttiAccess.....	80pp.	GetHashCode.....	89
RttiAttr.....	87	GetPackages.....	76
RttiAttrib.....	85p.	GetProcAddress.....	109
RttiIntro.....	65	GetTouchInputInfo.....	187
ShlObj.....	140	GetTypes.....	71p.
StopWatchTest.....	117	GetUserDefaultUILanguage.....	159
TiffViewer.....	154	GlassFrame.....	128
TouchMove.....	188, 190	Google.....	266, 270
TypeList.....	74	Google groups.....	10
TypesList.....	71, 73, 77	Gradients.....	149
Win7Libraries.....	140	GradientStartColor.....	202
Win7Taskbar.....	136, 139	Graphical Processing Unit.....	143
XmlPersist.....	93	Henri Gourvest.....	132
YahooMaps.....	268	Holger Flick.....	9
Execute.....	248p.	HP.....	166
Extended RTTI.....	64	HTML... 223, 225, 283p., 290, 296, 302, 304	
external viewer.....	54	HTTP.212pp., 221, 228, 232, 255, 260p., 285	
F6 key.....	23	HTTP Authentication.....	218
Fabrizio Schiavi.....	2	HTTP methods.....	292

312 - Index

HTTPAuthenticate.....	218	IOTANotifier interface.....	26
HTTPRIO.....	259	IOTAThreadNotifier.....	57p.
IAppServer.....	283	is.....	104
ICANN.....	280	ISAPI.....	222
ID2D1Brush.....	146	IShellItem.....	140pp.
ID2D1HwndRenderTarget.....	147	IShellLibrary.....	135, 140p.
ID2D1RenderTarget.....	144	ITaskbar.....	135
IDE.....	19	ITaskbarList.....	136
IDE Insight.....	23, 25	IWICBitmap.....	155
IdHTTP.....	261, 264, 286	IWICImagingFactory.....	155
IdHTTPServer.....	262, 281	Jaakko Salmenius.....	159
IDOMNodeSelect.....	265	JavaScript.....	233, 288p., 300, 304
IDWriteFactory.....	151	Jim Tierney.....	228
IFileDialog.....	128	Joint Endeavour of Delphi Innovators.....	132
IFileSaveDialog.....	128	jQuery.....	288p., 295p., 300pp., 305
IInertiaProcessor.....	191	JSON...233p., 240pp., 244p., 270pp., 284p.,	
ImageFormat.....	154	287, 292p., 302	
ImageList.....	138, 180	JSON Converters.....	240
IManipulationProcessor.....	191	JSON marshaling.....	293
Implicit.....	79	KeyCaptions.....	184
In-place Editor.....	203	Kylix.....	113
Incremental Search.....	29	Layout.....	184
indent.....	28	Lenovo.....	166
inertia.....	191, 194	LifeCycle.....	215
Inertia Manipulation.....	190	Live Templates.....	33
InitExceptions.....	107	LoadResourceModule.....	159
initialization.....	106	Local Variables.....	49
Input Language.....	158	localization support.....	113
Installation Folders.....	21	Location API.....	163
installation program.....	20	Lulu.....	7
InstallAware.....	19	MainFontOnTaskbar.....	127
Instruction Pointer.....	47	MainFormOnTaskbar.....	129
Int64.....	201	Malcolm Groves.....	85
INTAIDEInsightItem.....	26	manipulation.....	191
Integrated Development Environment.....	19	Manipulations.....	190
Interbase.....	208, 210	Map.....	268
interfacing .NET.....	131	Maps.....	268
Internal Translation Manager.....	159	Marco Breveglieri.....	9
Internet Engineering Task Force.....	233	Marco Cantù.....	9
Internet Explorer.....	290	Marshaling.....	237, 243
interposer class.....	204	Mastering Delphi.....	9
InvokeOption.....	259	Mastering Delphi 2005.....	256
IObjectArray.....	132	MDSN.....	188
IOTADebuggerServices.....	55	MessageFont.....	157
IOTADebuggerVisualizer.....	57	Microsoft SQL Server.....	210
IOTAIDEInsightService.....	25	Microsoft XML DOM.....	256

midas.dll.....	206	QueryPerformanceCounter.....	117
MonthCalendar.....	157	Recent Files.....	41
MSBUILD.....	34	Regional Settings.....	159
MSDN.....	126, 191, 193	RegisterTouchWindow.....	188
multi-threaded applications.....	50	RemoteServer.....	212
Multi-Touch.....	164, 186	RenderTarget.....	144, 152
Multiple Active Results Sets.....	211	Reopen.....	41
MySQL.....	210	Representational State Transfer.....	260
NameThreadForDebugging.....	52	Request.....	222
New file dialog box.....	43	Response.....	222
New Items dialog box.....	42	REST.....	212, 253, 255, 260p., 263, 274, 281, 302, 305
Object Inspector.....	36	Resume.....	114
OnAction.....	222p., 225, 275	RFC 4627.....	233
OnDrawDataCell.....	203	road map.....	5
OnExecute.....	174	Roy Fielding.....	260
OnGesture.....	168, 170, 172, 174, 178	RSS.....	263p.
OnMouseDown.....	189	RTTI.....	63
OnPaint.....	145, 151, 193	Run Time Type Information.....	63
OnTitleClick.....	202	SaveDialog.....	128
Open Tools API.....	25, 55	SAX.....	257pp.
Open XML.....	256	SAX interface.....	256
OpenDialog.....	128, 140	scoped enumerators.....	111, 119
Oracle.....	210	Screen.....	37, 157
OutputDebugString.....	49	SDK documentation.....	187
Overlay Icons.....	138	Search for forms.....	22
Packages.....	76	search panes.....	29
Paint.....	203	ServerClassName.....	212
ParamStr.....	68	SetOverlayIcon.....	138
ParseJSONValue.....	237	SetProgressState.....	137
PathInfo.....	280	ShellExecuteEx.....	130
Pawel Glowacki.....	151	ShellResources.....	128
Peter W A Wood.....	2	ShGetFolderPath.....	118p.
Peter Wood.....	8	SHGetKnownFolderItem.....	141
PopupActionBar.....	42	ShiftState.....	186
POST.....	298p.	Single.....	195
ProcessInput.....	231	SOAP.....	254, 259
ProcessOutput.....	231	Sony.....	166
ProgressBar.....	137, 157	Sort By.....	35
Project JEDI.....	132	SpeedButton.....	157
Project Manager.....	34p.	SQLConnection.....	212p., 221, 227, 243
Property Editor.....	157	SQLDataSet.....	220
PtInCircle.....	114	SqlServerMethod.....	212
PTypeInfo.....	67	square brackets.....	84
published.....	63	Standard.....	169
PUT.....	298p.	Standard Gestures.....	171
Quality Central.....	203		

314 - Index

Stefan Van As.....	9	TJSONMarshal.....	239
stored.....	97	TJSONNull.....	235
Synchronize.....	53	TJSONNumber.....	235, 245
Tab key.....	28	TJSONObject.....	235, 237p., 294
tablet support.....	133	TJSONPair.....	235
TArray.....	71	TJSONString.....	235
TArray<T>.....	114	TJSONTrue.....	235, 250
Taskbar Buttons.....	135	TJSONValue....	233, 235, 237pp., 241, 243p., 249
TaskDialog.....	128	TList.....	115
TaskMessageDlg.....	128	TObject.....	66
TCanvas.....	144	TObjectDictionary.....	293
TControlState.....	160	TObjectList.....	90
TCP/IP.....	212	Toolbar.....	181
TCustomAttribute.....	84, 89, 113	ToolsAPI.....	55
TCustomGrid.....	203	TopStyle4.....	9
TCustomIniFile.....	114	Touch.....	164, 168
TDateTime.....	53	Touch Hardware.....	165
TDBXCallback.....	248	Touch Keyboard.....	183
TDirect2DBrush.....	145p.	TouchInput.....	187
TDirect2DCanvas.....	144, 147, 149, 151	TouchKeyboard.....	183, 185
TDirect2DFont.....	145	TPath.....	118, 120
TDirect2DGraphicsObject.....	145	Transformations.....	154
TDirect2DPen.....	145	Translate API.....	270
TDirectory.....	118	TreeView.....	128
TEditButton.....	157	TRegistry.....	114
TEncoding.....	237	TRttiContext.....	65, 71pp., 76
Terminate.....	52	TRttiObject.....	72
TField.....	200	TRttiType.....	71
TFieldType.....	200	TShiftState.....	160, 186
TFile.....	118	TSingleField.....	200
TFilterPredicate.....	120p.	TSQLTimeStampOffsetField.....	200
TGestureCollectionItem.....	169	TStopWatch.....	117
The Delphi Magazine.....	204	TStreamReader.....	122
Themes.....	128	TStreamWriter.....	257
Thread Status View.....	51	TStringBuilder.....	114
Threads.....	50	TStringList.....	241
freezing.....	51	TStringWriter.....	278
thawing.....	51	TTextWriter.....	93, 278
ThumbBarAddButtons.....	139	TThread.....	52, 114
TIdEncoderMIME.....	231	TThumbButton.....	139
TIFF.....	153	TTimeSpan.....	117p.
TImage.....	154	TTouchManager.....	169
timer queue.....	133	TTouchMessage.....	189
TJSONArray.....	235, 243, 245, 295	TTransportFilter.....	231
TJSONConverter.....	237	TUncertainState.....	115
TJSONFalse.....	235, 248, 250		

- TValue.....78pp., 205
- TVisibilityClasses.....66, 113
- TWebAppSockObjectFactory.....224
- TWebModule.....222
- TWebRequest.....222
- TWebResponse.....222
- TWICImage.....132, 154pp.
- Twitter account.....10
- TypeInfo.....70
- UCS4Char.....56, 58
- unindent.....28
- unit.....
 - ADOMInt.....208
 - AdomCore_4_3.....256
 - Contrs.....116
 - Cor.....131
 - D2D1.....132, 151p.
 - DateUtils.....113
 - DB.....200
 - DBGrids.....204
 - DBXFirebird.....209
 - DBXFirebirdMetaData.....209
 - DBXJSON.....234pp., 287
 - DBXJSONReflect.....237
 - Diagnostics.....117
 - Direct2D.....115, 144, 151
 - Direct3D.....132
 - DSHTTPLayer.....219, 227
 - DwmApi.....134
 - DxgiFormat.....132
 - Generics.Collections.....293
 - Graphics.....144, 154
 - Grids.....203
 - IOUtils.....111, 117pp., 122
 - JPEG.....154
 - Keyboard.....183
 - KnownFolders.....140
 - Manipulations.....132, 191
 - Messages.....134
 - MsInkAut.....133
 - msxml.....256
 - mxsmll.....135
 - ObjectArray.....132
 - RtsCom.....133
 - Rtti.....64, 70pp., 78, 80, 82, 86, 117
 - ShellAPI.....135
 - SHFolder.....119
 - ShlObj.....118, 134, 136, 140
 - SqlTimSt.....200
 - StructuredQuery.....133
 - System.....66pp., 84, 105, 113p., 159
 - SysUtils.....68, 107, 115
 - TimeSpan.....117
 - ToolsAPI.....25, 55
 - TpcShrd.....133
 - Types.....114
 - TypeInfo.....70, 78, 95
 - Wincodec.....132
 - Windows.....68, 110, 133
 - WinSpool.....135
 - WMF9.....132
- UnregisterTouchWindow.....188
- URL.....260, 280
- Use Unit Dialog.....42
- Use Units dialog box.....22
- UseLatestCommonDialog.....140
- UseLatestCommonDialogs.....128
- User Account Control.....126
- value replacer.....54
- View Form.....22
- View Messages.....43
- View Units dialog box.....22
- Virtual Storage.....130
- Wacom.....166
- Watches.....49
- Web App Debugger 223p., 226, 275p., 284p., 292
- Web Services.....254
- WebBroker.....212, 221pp., 282, 292
- WebModule.....222
- Win32MajorVersion.....126
- Windows 7.....126, 135, 165, 191
- Windows API.....126
- Windows CE.....164
- Windows Display Driver Model.....143
- Windows Imaging Component.....132, 153
- Windows Presentation Foundation.....143
- Windows Search.....133
- Windows Search SDK.....133
- wm_ClipboardUpdate.....134
- wm_Gesture.....134
- wm_InputLangChange.....158

316 - Index

wm_InputLanguageChange.....	185	XPath.....	262, 264
wm_SysCommand.....	139	Yahoo.....	268
wm_touch.....	134, 165, 186, 189	\$RTTI.....	66
XML.....	93, 99, 255, 259p., 262, 276	\$StrongLinkTypes.....	70
XmlData.....	257	\$WeakLinkRTTI.....	70
XMLDocument.....	255, 257, 262, 264, 280		

Advertisers Index

This is the list of third-party component vendors, database vendors, and partners working with the Delphi community, who are advertising in this book.

- Developer Express, page 17 and 123
- Raize Software, page 45
- Steema Software, page 61
- Gnostice, page 101
- Delphi Developer Days, page 161
- Advantage Database Server, page 198

Web Sites by Marco Cantù

Here is a partial list of the diverse and somewhat unrelated web sites I manage (or don't manage enough, as some of them are quite old and static) in English language:

- <http://www.marcocantu.com>
- <http://blog.marcocantu.com>
- <http://www.thedelphisearch.com>
- <http://www.wintech-italia.com>
- <http://dev.newswat.com>
- <http://delphi.newswat.com>
- <http://ajax.marcocantu.com>
- <http://www.delphimentor.com>
- <http://www.socialwebbook.com>

Here are other sites in Italian language:

- <http://www.marcocantu.it>
- <http://www.wintech-italia.it>
- <http://shop.wintech-italia.com>
- <http://www.delphieditorni.it>
- <http://www.piazzacavalli.net>

Personal pages on community sites and micro-blogging sites:

- <http://twitter.com/marcocantu>
- <http://www.facebook.com/marcocantu>
- <http://www.linkedin.com/in/marcocantu>
- <http://www.librarything.com/profile/MarcoCantu>
- <http://marcocantu.mylaxo.com/>

My online shops (where you can buy books, tools, and services) include:

- <http://sites.fastspring.com/wintechitalia>
- <http://blog.marcocantu.com/bookstore.html>
- <http://shop.wintech-italia.com> (Italian)

